Titre: Title:	Algorithme de maintien de cohérence dans les bases de données réparties sur grappes d'ordinateurs
Auteur: Author:	Constant Wette Tchouati
Date:	2000
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Wette Tchouati, C. (2000). Algorithme de maintien de cohérence dans les bases de données réparties sur grappes d'ordinateurs [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8893/

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ALGORITHME DE MAINTIEN DE COHÉRENCE DANS LES BASES DE DONNÉES RÉPARTIES SUR GRAPPES D'ORDINATEURS

présenté par : WETTE TCHOUATI Constant

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été accepté par le jury d'examen composé de :

Martine Bellaïche, M.Sc.	Président
Samuel Pierre, Ph.D.	Directeur
Jean Conan, Ph.D.	Codirecteur
Alejandro Quintero, Ph.D.	Membre

NOTE TO USERS

Page(s) not included in the original manuscript are unavailable from the author or university. The manuscript was microfilmed as received.

iii

This reproduction is the best copy available.

REMERCIEMENTS

À M. Samuel Pierre dont le soutien et les conseils m'ont guidé tout au long de ma recherche

À M. Jean Conan dont les conseils m'ont guidé tout au long de mon cycle de maîtrise

À mes parents, frère et sœurs au Cameroun pour leur soutien constant et indéfectible

À Luc Parent, Ibrahim Heddad, Makan Poudzandi, Per Anderson pour leur disponibilité et leur collaboration pendant mes travaux à Ericsson

À tous mes collègues du Laboratoire de Recherche en Réseautique et Informatique Mobile (LARIM) pour leur collaboration et surtout pour l'ambiance de travail chaleureuse.

RÉSUMÉ

Durant les deux dernières décennies, le domaine des bases de données (BD) a connu beaucoup de changements avec le développement des réseaux téléinformatiques. L'introduction des concepts de répartition et de réplication des données sur différents sites a eu pour conséquence de compliquer la conception des algorithmes implantés dans les systèmes de gestion de bases de données répartis (SGBDR) par rapport à leurs équivalents centralisés. Un algorithme de contrôle des accès concurrents est une solution au problème de maintien de cohérence dans une base de données et constitue l'objet de notre étude. Cet algorithme a pour fonction d'ordonnancer localement et globalement les opérations des transactions concurrentes de manière à garantir la cohérence de la BD répartie à la fin des exécutions.

On distingue trois types d'algorithmes de contrôle des accès concurrents : l'algorithme de verrouillage, l'algorithme d'estampillage et l'algorithme optimiste. Dans une BD répartie, une entité de donnée est partagée non seulement par les applications du même site, mais aussi par des applications des sites distants. Cependant, la contrainte de maintien de cohérence doit être satisfaite par l'algorithme; ainsi, lorsque le degré de concurrence augmente, des transactions sont retardées dans le cas de l'algorithme de verrouillage ou réinitialisées dans le cas de l'algorithme optimiste. Le temps de réponse moyen du système devient alors plus important et entraîne une dégradation de performance. Dans les systèmes réels, pour limiter ce phénomène, les concepteurs des algorithmes de contrôle des accès concurrents font plutôt un compromis en relaxant un peu la contrainte de maintien de cohérence pour permettre un niveau de performance acceptable.

Ce mémoire a pour objectif principal de concevoir un algorithme de maintien de cohérence pour les bases de données des grappes d'ordinateurs, en tenant compte des exigences de haute disponibilité et de haute performance de certains types d'applications. Pour y parvenir, nous avons d'abord recensé et analysé les différents algorithmes qui sont implantés dans les SGBDs commerciaux, ainsi que celui qui est

utilisé dans DBN du système TelORB d'Ericsson. Tous ces algorithmes utilisent la méthode de verrouillage qui est souvent qualifiée de pessimiste puisqu'elle vérifie les conflits entre transactions avant leur phase d'exécution. Nous avons aussi étudié des algorithmes dits optimistes où la vérification de conflits se fait après la phase d'exécution, mais ils sont encore au stade de prototypage sans être encore implantés dans un système réel.

De tous les algorithmes explorés et d'après un certain nombre de critères que nous avons définis, nous avons sélectionné un algorithme de verrouillage et un algorithme optimiste que nous avons codés. Nous avons également proposé un nouvel algorithme dont l'implémentation nous a permis de mesurer et de comparer sa performance avec les deux algorithmes précédents. Nous avons testé chacun des trois algorithmes sous différentes configurations possibles : SGBDR en mémoire vive, SGBDR sur disque, SGBDR avec réplication de données, SGBDR de temps réel. En variant le taux d'arrivée des transactions et leur degré de complexité, nous avons mesuré les indices de performance tel que le débit (taux de transactions complétées) du système et le temps de réponse moyen.

Des résultats obtenus, nous constatons qu'en présence de conflits entre transactions, l'algorithme de verrouillage offre de meilleures performances par rapport à la méthode optimiste, tandis qu'en absence de conflits c'est plutôt l'algorithme optimiste qui fournit de meilleurs résultats. L'algorithme que nous proposons offre de bonnes performances en présence et en absence de conflits. Dans une base de données répartie, la probabilité de conflits entre un groupe d'applications qui partagent des données pourrait être estimée à l'avance de manière à adopter un contrôle d'accès pessimiste ou optimiste. Or, les exigences des applications sont parfois contradictoires. Ainsi, un SGBD qui applique une seule méthode de contrôle d'accès ne saurait répondre à toutes les situations. Si les trois algorithmes sont implantés dans un SGBD, notre algorithme offrirait la meilleure garantie de performance par rapport aux deux autres, pour différents degrés de conflit entre transactions.

ABSTRACT

During the two last decades, the development of computer networks brought numerous changes in the design of database systems. The introduction of the concepts of data distribution and replication on various sites, results in to complicate the design of algorithms for distributed database management systems (DDBMS) compared to their equivalent for a centralized DBMS. A concurrency control algorithm solves the problem of coherence preservation in a database and constitutes the object of our study. The purpose of that algorithm is to produce a local and a global schedule on the operations of concurrent transactions so as to preserve the coherence of the database at the end of execution.

Three basic concurrency control algorithms have been designed to solve the problem of coherence preservation: locking, timestamp and the optimistic algorithms. In a distributed database, a data entity is shared not only by the applications of the same site, but also by applications of remote sites. However, the algorithm must satisfy the constraint of coherence preservation. When the level of contention increases, some transactions are either delayed in the case of the locking, or abort in the case of the optimistic algorithm, and this situation leads to degradation of performance. To limit this phenomenon, designers of concurrency control algorithms use to make a trade-off by releasing some constraints of coherence preservation to achieve an acceptable level of performance.

Our main objective in this project is to design a suitable concurrency control algorithm for distributed databases on computer clusters that also meets the requirements of high availability and high performance, necessary to certain types of applications. For that purpose, we analyzed some algorithms that are implemented in commercial DBMSs, as well as the algorithm in Ericsson operating system TelORB/DBN. All these algorithms use the locking method, which is often described as pessimistic since it checks conflicts between transactions before their execution phase. We also studied optimistic algorithms where conflicts are checked after the execution

phase, but they are still at the stage of prototyping without being implemented in a real system. From all the explored algorithms and considering a set of criteria, we selected one locking algorithm and one optimistic algorithm which we coded together with a new algorithm that we propose in a simulator who allowed us to measure and to compare their performance. We tested each of the three algorithms under various configurations: in-memory DBMS, disk-based DBMS, replicated DBMS and real-time DBMS. By varying the arrival rate of transactions and their level of complexity, we measure the throughput (number of transactions completed per second) of the system and the average response time.

Experimental results shown that when transactions conflict, the locking algorithm performs better compared to optimistic, while optimistic algorithm produces the best performance when conflicts are very rare. The algorithm we propose still offers good performance when transactions conflict or not. In a distributed database, the probability of conflicts between groups of applications that share some data could be considered in advance to adopt either a pessimistic or optimistic concurrency control. However, applications requirements are sometimes contradictory, thus a DBMS which applies only one concurrency control method will not be suitable for all the situations. If the three algorithms are implemented in a DBMS, our algorithm is guaranteed to offer best performance for different levels of transactions conflict compared to the two other algorithms.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

Sigle ou abréviation	Signification
ACID	Atomicité - Cohérence - Isolation - Durabilité
BD	Bases de Données
BDR	Bases de Données Réparties
CORBA	Common Object Request Broker Architecture
DBN	SGBD intégré dans TelORB
GD	Gestionnaire de Données
GT	Gestionnaire des Transactions
GSM	Global System for Mobile
JDBC	Java Database Connectivity
2PL	Two Phase Locking
HLR/VLR	Home Location Register / Visitor Location Register
OCC	Optimistic Concurrency Control
ODBC	Open Database Connectivity
ODMG	Object Database Management Group
OPT	Ordonnanceur Optimiste
OQL	Object Query Language
OR	Ordonnanceur
SCP	Service Control Point (SS7)
SGBD	Système de Gestion des Bases de Données
SGBDR	Système de Gestion de Bases de Données Réparties
SQL	Structured Query Language
SS7	Signaling System 7
TelORB	Système d'exploitation propriétaire d'Ericsson
V2P	Verrouillage en deux Phases
XML	Extensible Markup Language

CHAPITRE I

INTRODUCTION

Les bases de données sont utilisées dans de nombreuses applications informatiques, allant de la météo au commerce électronique. Durant les deux dernières décennies, le développement des réseaux téléinformatiques et des systèmes répartis a introduit dans la communauté des chercheurs en bases de données l'important concept de bases de données réparties. La répartition et la réplication des données sur différents sites ont pour principaux avantages : la décentralisation de la gestion des données, la haute disponibilité des données vis-à-vis des applications et la possibilité d'exécuter des requêtes en parallèle. Cependant, la contrainte de maintien de cohérence des données lorsqu'elles sont réparties ou répliquées rend plus complexe la conception des diverses composantes d'un système de gestion de bases de données (SGBD). Ce mémoire traite des modèles de gestion des accès concurrents dans une base de données répartie avec garantie de performance. Dans ce chapitre d'introduction, nous définirons d'abord quelques concepts de base et les éléments de la problématique, ensuite nous préciserons nos objectifs de recherche et résumerons les résultats attendus, enfin nous esquisserons les grandes lignes du mémoire.

1.1 Définitions et concepts de base

Une *base de données* (BD) est une collection de données enregistrées ensemble, sans redondance pénible ou inutile, pour servir plusieurs applications; on y enregistre les données de façon à ce qu'elles soient indépendantes des programmes qui les utilisent. Un système de gestion de base de données (SGBD) est un logiciel qui sert à créer des bases de données; il permet aussi d'ajouter, d'interroger ou de modifier des données

stockées en mémoires secondaires. À chaque instant, l'état de la base de données est caractérisé par les valeurs des objets qu'elle contient et qui représentent de façon *cohérente* une partie du monde réel. Cet état est modifié par l'exécution d'une transaction. Une *transaction* est un programme fiable constitué d'un ensemble d'opérations dont l'exécution a pour conséquence de faire passer la BD d'un état cohérent à un autre état cohérent. L'état cohérent de la base de données est défini par un ensemble de règles mémorisées par le SGBD appelé *contraintes d'intégrité*. La cohérence et la fiabilité d'une transaction est une conséquence des quatre propriétés fondamentales ACID qui correspondent aux quatre contraintes suivantes :

- ***atomicité*** : le système doit valider *toutes* les mises à jour effectuées par une transaction ou ne *rien* faire du tout;
- ***cohérence*** : en cas d'échec d'une transaction, l'état cohérent initial doit être restauré;
- ***isolation*** : les résultats des mises à jour effectuées par une transaction ne doivent être visibles aux autres transactions qu'une fois la transaction validée, afin d'éviter les interférences;
- ***durabilité*** : dès qu'une transaction valide ses modifications, le système doit garantir qu'elles seront conservées, même en cas de panne.

Ces propriétés doivent être garanties dans le cadre d'un SGBD centralisé, mais aussi dans les systèmes répartis. Elles nécessitent deux types de contrôle, qui sont intégrés dans un SGBD : contrôle de concurrence et résistance aux pannes avec validation de reprise.

Les principales composantes d'un SGBD réparti sont représentées à la Figure 1.1. Chaque rangée représente un site qui comprend : un gestionnaire des transactions (GT), un ordonnanceur (OR), un gestionnaire de données (GD) et une base de données. Le réseau de communications est matérialisé par l'ensemble des liaisons qui relient des composantes de sites différents. L'*ordonnanceur* est un programme qui a pour fonction principale de gérer ou de synchroniser les accès concurrents afin d'éviter les incohérences.

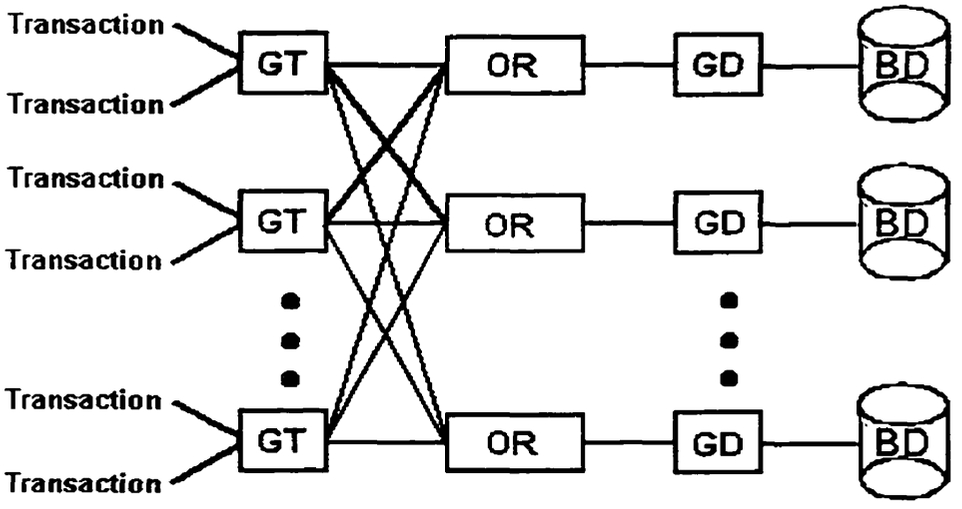


Figure 1.1 Architecture d'un SGBD réparti

Une *base de données répartie* (BDR) est un ensemble de bases de données stockées sur différents sites qui sont reliés par un réseau de communications et qui apparaissent aux applications comme une seule base de données. Dans un environnement réparti, une transaction qui accède à un ensemble de données stockées sur des sites différents est appelée *transaction répartie*. Un site reçoit une transaction répartie, la décompose en *sous-transactions* puis assure la coordination de leur exécution sur les sites qui abritent les replicas de données accédées. Par convention, la répartition et la réplication des données doivent être *transparentes* aux applications. Par exemple, la BD peut être modélisée comme un ensemble d'entités logiques qui peuvent avoir une ou plusieurs copies physiques qui représentent ses *replicas*. Les applications ont une vue des données logiques; lors des accès, le SGBD est responsable de la traduction d'une part d'une opération de lecture sur une donnée logique en une opération de lecture sur un seul de ses replicas, et d'autre part d'une opération d'écriture sur une donnée logique en opérations d'écriture sur chacun des replicas. Cet exemple illustre en fait le principe d'un protocole bien connu appelé *ROWA (Read Once Write*

All) qui est présenté dans (Helal, 1997). Très souvent, la *répartition* des données sur différents sites est effectuée de façon à minimiser le coût de gestion et les risques de défaillance du système par rapport à l'approche centralisée. La *réplication*, quant à elle, consiste à stocker sur plusieurs sites, des copies d'une même donnée logique fournissant ainsi une *haute disponibilité* des données. Afin de garantir la cohérence des données, la mise à jour des différents réplicas doit être synchronisée, ce qui augmente le temps de reponse du système.

Dans les SGBDs, le disque représente la mémoire stable et il permet de mémoriser les données persistantes. Afin d'améliorer le temps de réponse dans certaines applications, le SGBD peut utiliser l'antémémoire ou encore la base de données peut être stockée sur mémoire volatile. Dans ce dernier cas, la persistance est rendue possible grâce au mécanisme de réplication. À un instant donné, l'état de la base de donnée est déterminé par l'état de la mémoire stable et l'état de l'antémémoire. En effet, des mises à jour ont été effectuées et ne sont pas encore reportées sur disque. Certaines d'entre elles effectuées par des transactions venant juste d'être validées peuvent être en cours de report. Il faut cependant garantir la non-perte de mise à jour des transactions commises en cas de panne. Si le système reporte des pages dans la mémoire stable avant validation (*commit*) d'une transaction, il faut être capable de défaire (*undo*) les reports des pages contenant des mises à jour de transactions annulées (*abort*).

La méthode la plus classique pour permettre la validation atomique, l'annulation et la reprise de transaction consiste à utiliser des journaux. Un *journal* est un fichier système qui contient les valeurs des pages mises à jour, dans l'ordre des modifications avec les identifiants des transactions modifiantes, ainsi que des enregistrements indiquant les débuts, validation et annulation de transactions. On distingue deux sortes de journaux : le *journal des images avant* qui est utilisé pour défaire les mises à jour d'une transaction annulée, et le *journal des images après* qui est utilisé pour refaire (*redo*) les mises à jour d'une transaction validée en cas de panne.

1.2 Éléments de la problématique

Une BD répartie est gérée par un SGBD réparti dont les composantes sont installées sur différents sites, comme illustré à la Figure 1.1. Examinons le scénario de communications entre les différentes composantes pendant l'exécution d'une transaction dans le cas le plus général où les données sont répliquées. Une transaction émise par une application est enregistrée par le GT avec un identificateur TrId. Ainsi, pour chaque opération, le GT contacte l'ordonnanceur (OR) qui contrôle les accès à la donnée sollicitée. S'il s'agit d'une opération de lecture, l'OR contacte un seul des sites où se trouve un réplica de la donnée; mais lorsqu'il s'agit d'une opération d'écriture, chacun des sites où se trouve un réplica doit être contacté. Les ORs sont chargés non seulement de synchroniser les différents accès aux données, mais aussi de produire un ordonnancement globalement sérialisable pour préserver la cohérence de la BDR après exécution des transactions en cours. Pour y parvenir, l'OR peut soit envoyer directement une opération au GD lorsqu'il n'y a pas de conflit, la retarder ou la rejeter en cas de conflit. Et dans le cas d'un rejet, la transaction est réinitialisée (*restart*) par l'application. Le GD à son tour exécute tout simplement les opérations qu'il reçoit sans plus se soucier de les ordonnancer. S'il s'agit d'une lecture, le GD retourne la valeur de l'entité; s'il s'agit d'une écriture, le GD modifie sa valeur locale et retourne à l'OR un acquittement que celui-ci relaie au GT qui le relaie ensuite au programme qui a émis la transaction.

Les mécanismes d'ordonnancement des transactions peuvent être regroupés en trois principales classes : verrouillage, estampillage et méthode dite optimiste. Le verrouillage et l'estampillage résolvent d'abord les conflits avant d'exécuter les transactions en se basant sur les informations syntaxiques. Ce principe garantit a priori la cohérence des données après exécution des opérations des transactions, mais le temps de réponse peut être assez important. La méthode optimiste, quant à elle, suppose que les conflits entre transactions ne sont pas fréquents et tolère les accès multiples à une même donnée, puis procède à la validation. Contrairement aux deux autres méthodes

qui sont dites pessimistes, la résolution des conflits après exécution tient surtout compte de l'information sémantique. En cas de conflit, une transaction peut être réinitialisée.

Des dizaines d'algorithmes optimistes sont proposés et les travaux réalisés jusqu'à nos jours démontrent que la méthode optimiste offre une meilleure performance par rapport au verrouillage lorsqu'il y a moins de conflits ou encore lorsque le taux d'arrivée des transactions est élevé. Cependant, en utilisant l'approche optimiste, beaucoup de transactions sont réinitialisées lorsque les conflits deviennent fréquents, et la performance a plutôt tendance à se dégrader en comparaison avec le verrouillage. C'est ainsi que le verrouillage reste encore la méthode la plus utilisée dans les SGBD commerciaux.

Dans chacune des deux méthodes, les objectifs visés consistent d'une part à minimiser le nombre de messages échangés et à répartir leur traitement uniformément entre les processeurs, puis d'autre part à minimiser la quantité de données stockées temporairement en utilisant uniformément la mémoire des processeurs. L'efficacité d'une méthode d'ordonnancement dépend alors des exigences du service offert et de la configuration du matériel utilisé. En particulier dans les grappes (clusters) d'ordinateurs, les exigences à satisfaire vis-à-vis des usagers (transactions) sont les suivantes : temps de réponse minimum, haute disponibilité des données, contraintes temps réel, haut débit (high throughput), garantie de cohérence des données. Par ailleurs, une grappe d'ordinateurs n'est pas nécessairement constituée de matériel haut de gamme (processeur, réseau de communications, mémoire de stockage) et doit satisfaire aussi la contrainte d'évolutivité (scalability). Pour répondre aux besoins sans cesse croissants des usagers, il est nécessaire d'explorer davantage les trois méthodes classiques d'ordonnancement pour dériver un nouveau mécanisme qui s'adapte bien aux exigences des clusters.

1.3 Objectifs de recherche

L'objectif principal de ce mémoire est de concevoir un algorithme de maintien

de cohérence dans les bases de données des grappes d'ordinateurs en tenant compte des exigences de haute disponibilité et de haute performance de certains types d'applications. De manière plus spécifique, ce mémoire vise à :

- analyser les phénomènes qui peuvent compromettre la cohérence des bases de données réparties sur des grappes d'ordinateurs, ainsi que les solutions qui y ont été proposées;
- proposer un algorithme général pour améliorer la performance et la capacité de maintien de cohérence des systèmes répartis sur des plates-formes reconfigurables comme les grappes d'ordinateurs;
- évaluer l'efficacité de l'algorithme proposé, en regard des meilleurs algorithmes existants dans la littérature, en particulier celui implanté dans le système TelORB d'Ericsson.

1.4 Plan du mémoire

Ce mémoire comprend 6 chapitres. Le chapitre suivant présente une analyse des phénomènes qui affectent la cohérence et les différents approches et algorithmes qui sont utilisés pour résoudre ce problème. Le chapitre 3 décrit l'approche que nous proposons pour résoudre ce problème dans un système réparti. Le chapitre 4 présente les détails d'implémentation de notre algorithme de gestion d'accès concurrents. Le chapitre 5 présente une analyse détaillée des résultats fournis par l'implémentation, assortie de comparaisons avec d'autres méthodes explorées. Enfin, le chapitre 6 résume les principaux résultats obtenus, les limitations de la méthode proposée et les extensions possibles aux travaux déjà entrepris.

CHAPITRE II
MODÈLES ET ALGORITHMES DE MAINTIEN DE COHÉRENCE

De façon générale, les situations qui peuvent entraîner des incohérences dans une base de données sont : les accès concurrents, les pannes qui surviennent au cours de la mise à jour des données et la validation des transactions incorrectes. Pour chacune des trois situations, un SGBD intègre soit un mécanisme de prévention, soit un mécanisme de correction, afin de garantir la cohérence des données lorsqu'elles sont accédées par des transactions. Dans ce chapitre, nous passons en revue les modèles et algorithmes de gestion des accès concurrents dans les bases de données réparties (BDR). Nous définirons d'abord quelques concepts de base, ensuite nous exposerons la théorie qui régit les mécanismes de contrôle des accès concurrents dans les BDs, puis nous présenterons les différents algorithmes qui sont utilisés pour les implanter dans les SGBDs.

2.1 Définitions et concepts de base

Dans une BD, des transactions qui accèdent simultanément à la même entité de donnée peuvent entraîner des pertes de mises à jour ou encore l'apparition des incohérences. Un *ordonnanceur* est un programme qui est utilisé pour synchroniser les accès concurrents afin de garantir la cohérence et l'isolation des mises à jour des transactions (le C et le I de ACID). Pour garantir la résistance aux pannes, les mises à jour sont effectuées à l'aide d'un protocole fiable dite de validation en deux phases (*Two Phase Commit*) qui permet aussi d'assurer l'atomicité et la durabilité des mises à jour des transactions (le A et le D de ACID). Par ailleurs, les données d'une BD ne sont pas indépendantes mais obéissent à des règles sémantiques appelées *contraintes d'intégrité*. Ces règles sont déclarées et mémorisées dans le dictionnaire de données. Ainsi, lors

d'une validation, le SGBD vérifie que ces contraintes ne sont pas violées par les mises à jour effectuées par une transaction. Notre étude porte sur la gestion des accès concurrents. Comme hypothèses, nous supposons que la couche système d'exploitation et la couche réseau sur lesquelles est installé le SGBD sont fiables et que les transactions que celui-ci reçoit sont des programmes corrects qui ne violent pas les contraintes d'intégrité.

Une *perte de mise à jour* survient lorsqu'une transaction T_1 exécute une mise à jour calculée à partir d'une valeur périmée de donnée, c'est-à-dire d'une valeur modifiée par une autre transaction T_2 depuis la lecture par la transaction T_1. La mise à jour de T_2 est donc écrasée par celle de T_1. Une perte de mise à jour est illustrée à la Figure 2.1.

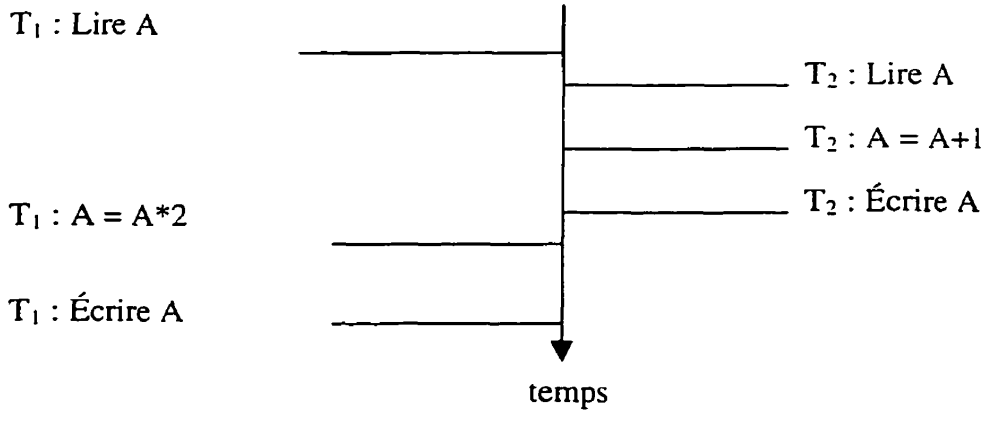


Figure 2.1 Exemple de perte de mise à jour

Une *incohérence* apparaît lorsque des données liées par une contrainte d'intégrité sont mises à jour par deux transactions dans des ordres différents qui amènent à violer la contrainte. Pour illustrer, considérons deux réplicas A et B devant rester égales. L'exécution des deux sequences d'opérations { T_1 : A=A+1 : T_2 : A=A*2 } et { T_2 : B=B*2; T_1 : B=B+1} rend en général A différent de B. Elle provoque donc l'apparition d'une incohérence. Cette situation est illustrée à la Figure 2.2.

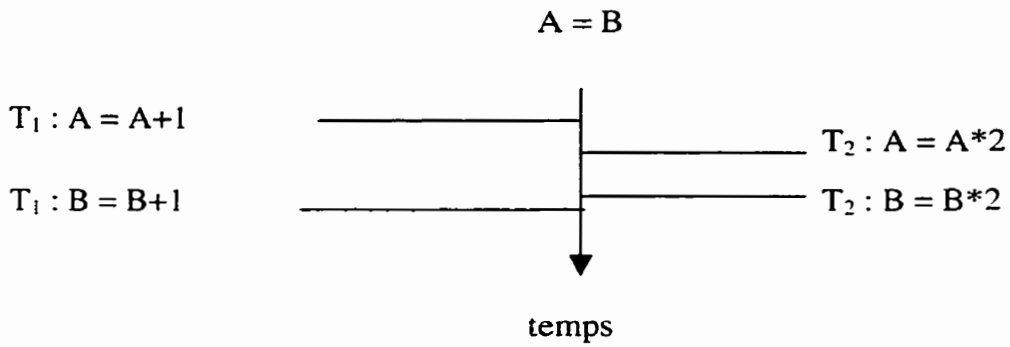


Figure 2.2 Exemple d'introduction d'incohérence

Un autre problème lié aux accès concurrents est la *non-reproductibilité des lectures* : deux lectures d'une même donnée dans une même transaction peuvent conduire à des valeurs differentes si la donnée est modifiée par une autre transaction entre les deux lectures. La Figure 2.3 en est une illustration.

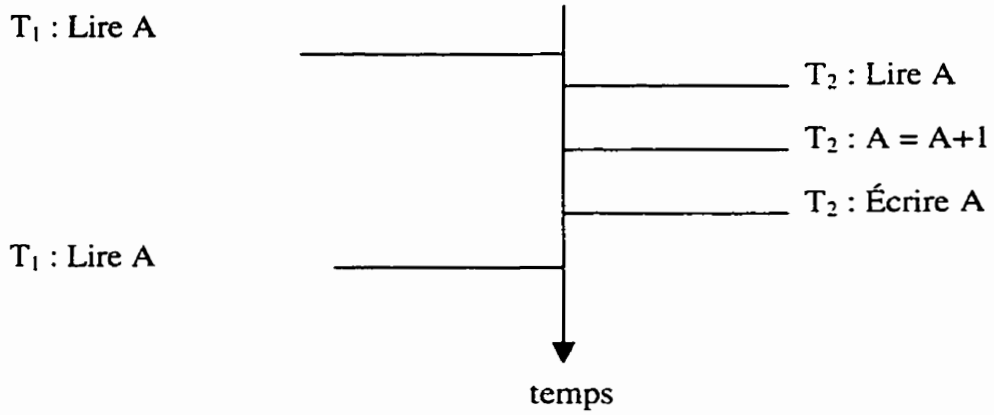


Figure 2.3 Exemple de non-reproductibilité des lectures

On peut éviter les problèmes d'incohérence ou de non-reproductibilité des lectures en isolant les mises à jour, c'est-à-dire en rendant les modifications invisibles à une autre transaction avant la fin de la transaction. Quant aux pertes de mise à jour,

l'isolation des mises à jour n'est pas suffisante, il faut aussi empêcher que deux transactions modifient simultanément une même donnée. Dans un SGBD, la résolution des problèmes évoqués ci-dessus nécessite la mise en place d'algorithmes de contrôle d'accès concurrents. Ces algorithmes s'appuient sur la théorie de la sérialisabilité, que nous examinons dans la section suivante.

Un *granule de concurrence* est une unité de données dont les accès sont contrôlés individuellement par le SGBD. Un granule peut être une ligne, une page ou une table dans un système relationnel, il peut être un objet ou une page dans un SGBD objet. Une *action* est une unité indivisible exécutée par le SGBD sur un granule pour une transaction, elle est généralement constituée par une lecture ou une écriture. Une *opération* est une suite d'actions accomplissant une fonction sur un granule en respectant sa cohérence interne : lecture, écriture, modification, insertion, suppression. Deux opérations sont en *conflit* si elles tentent d'accéder simultanément à la même granule. Un *ordonnancement* est une suite d'opérations pouvant représenter une histoire possible des exécutions effectuées par le système pour un certain nombre de transactions.

2.2 Théorie de la sérialisabilité

La théorie de la sérialisabilité (Papadimitriou, 1986) est un ensemble de règles mathématiques qui sont utilisées pour valider un algorithme de gestion des accès concurrents (ou d'ordonnancement des transactions). Un tel algorithme, encore appelé *ordonnanceur*, est correct s'il transforme tout ensemble de transactions concurrentes en un ordonnancement qui garantit la cohérence de la base après son exécution.

Notations

Une opération de lecture ou d'écriture d'une transaction T_i sur une donnée x est notée $O_i(x)$. $R_i(x)$ et $W_i(x)$ désignent respectivement une opération de lecture ou d'écriture de T_i sur une donnée x. Une transaction est modélisée par $T_i = (\sum_i, \alpha_i)$ où $\sum_i =$

{ $O_i(x)$ / O_i est une opération effectuée sur une donnée x pour la transaction T_i }, et α_i l'ordre dans lequel ces opérations doivent être effectuées. Soit $T = \{T_1, T_2, ... T_n\}$ un ensemble de transactions soumises à un SGBD réparti. Un ordonnancement complet S^C défini sur T [Özsu et Valduriez, 1999] est un ordre partiel $S^C = \{\Sigma, \alpha\}$ où :

1- $\Sigma = \cup^n_{i=1} \Sigma_i$

2- $\alpha \supseteq \cup^n_{i=1} \alpha_i$

3- si deux opérations $O_i, O_j \in \Sigma$ sont en conflits soit $O_i \alpha O_j$, soit $O_j \alpha O_i$ (2.1)

La première condition stipule que le domaine de l'ordonnancement Σ est égal à l'union des domaines Σ_i des T_i. La relation d'ordre α est le majorant des relations d'ordre α_i des T_i, cette deuxième condition signifie que l'ordonnancent global des opérations respecte l'ordre établi entre les opérations pour chaque transaction pris individuellement. Un *ordonnancement* est en effet un préfixe d'un ordonnancement complet. Un ordonnancement complet défini l'ordre d'exécution pour toutes les opérations de son domaine, alors qu'un ordonnancement peut être restreint sur une partie seulement des opérations des différentes transactions, plus particulièrement celles qui sont en conflits. $S = \{\Sigma', \alpha'\}$ est préfixe d'un ordre partiel $S^C = \{\Sigma,\alpha\}$ si :

1- $\Sigma' \subset \Sigma$

2- $\forall e_i \in \Sigma'$, $e_1 \alpha' e_2$ si et seulement si $e_1 \alpha e_2$

3- $\forall e_i \in \Sigma'$, si $\exists e_j \in \Sigma$ et $e_j \alpha e_i$, alors $e_j \in \Sigma'$.

Exemple 2.1 : Soit un ensemble de transactions concurrentes $T = \{T_1, T_2, T_3, T_4\}$ où $T_1 = W_1[x]W_1[y]W_1[z]$; $T_2 = R_2[x]R_2[z]W_2[x]$; $T_3 = R_3[x]W_3[y]$; $T_4 = R_4[z]W_4[y]W_4[z]$

Un ordonnancement possible S_0 sur T est schématisé à la Figure 2.4.

$S_0 = W_1[x]W_1[y]W_1[z]\ R_3[x]W_3[y]\ R_2[x]R_2[z]W_2[x]\ R_4[z]W_4[y]W_4[z]$

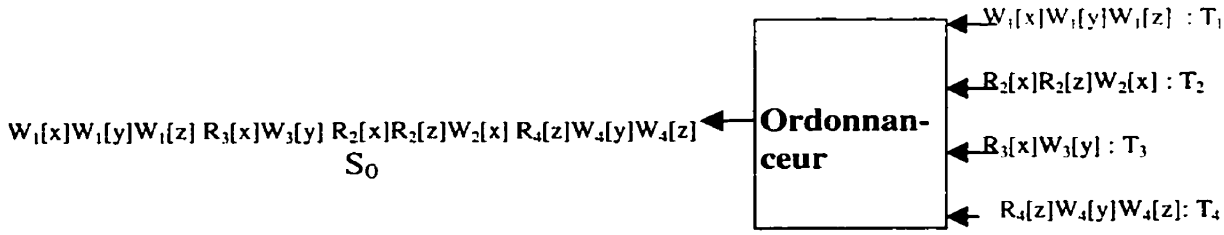


Figure 2.4 Exemple d'ordonnancement

Equivalence entre deux ordonnancements

Soient $W_i[x]$ et $R_j[x]$ deux operations contenues dans S, alors $R_j[x]$ ***lit-de*** $W_i[x]$ si $W_i[x]$ α $R_j[x]$ et il n'existe pas de $W_k[x]$ entre $R_j[x]$ et $W_i[x]$.

Exemple 2.2 : Dans l'ordonnancement $W_0[x] \rightarrow R_1[x] \rightarrow W_2[x] \rightarrow R_3[x] \rightarrow R_4[x]$;
$R_1[x]$ lit-de $W_0[x]$ alors que $R_3[x]$ ou $R_4[x]$ lit-de $W_2[x]$

$W_i[x]$ est appelée ***écriture finale*** s'il n'est suivi par aucune opération $W_k[x]$

Exemple 2.3 : Dans l'ordonnancement $W_0[x] \rightarrow W_1[x] \rightarrow W_2[x] \rightarrow R_2[x]$
$W_1[x]$ et $W_2[x]$ sont des écritures finales.

Intuitivement, deux ordonnancement S_1 et S_2 définis sur le même ensemble de transactions $S = \{T_1, T_2,, \ldots T_n\}$ sont dits ***équivalents*** s'ils transforment un état initial donné de la base en un même état final. C'est-à- dire :

1- chaque opération $R_j[x]$ lit-du même $W_i[x]$ dans S_1 et S_2

2- S_1 et S_2 ont les mêmes écritures finales.

Exemple 2.4 :

a) Un ordonnancement équivalent à S_0 de l'exemple 2.1 serait :

$S_1 = W_1[x]\ R_2[x]\ R_3[x]\ W_2[x]\ W_1[y]\ W_3[y]\ W_1[z]\ R_2[z]\ R_4[z] W_4[y] W_4[z]$.

b) un ordonnancement non-équivalent à S_0 serait :

$S_2 = W_1[x]\ R_2[x]\ R_3[x]\ W_2[x]\ W_1[y]\ W_1[z]\ R_2[z]\ R_4[z] W_4[y]\ W_3[y]\ W_4[z]$, car ils n'ont pas les mêmes écriture finales : $W_4[y]$ et $W_3[y]$ ne sont pas exécutés dans le même ordre.

Ordonnancement sérialisable

Une transaction est un programme correct qui transforme la BD d'un état cohérent à un autre état cohérent. Un ordonnancement *S* de transactions est dit *sériel* si les différentes opérations des transactions qui la composent ne s'interfèrent pas. Un ordonnancement sériel est supposé correct car, à chaque instant, une seule transaction est exécutée à la fois.

Exemple 2.5 : L'ordonnancement S_0 de l'exemple 2.1 est sériel.

Un ordonnancement S est dit *sérialisable* s'il existe un ordonnancement sériel S_0 tel que S est équivalent à S_0.

Exemple 2.6 : L'ordonnancement S_1 est sérialisable car il est équivalent à S_0. S_2 n'est équivalent à aucun ordonnancement sériel, donc S_2 n'est pas sérialisable.

Remarque : Tout ordonnancement sérialisable est supposé correct.

Un algorithme d'ordonnancement est dit *correct* s'il ne produit que des ordonnancements sérialisables. Le critère de sérialisabilité doit être vérifié si le SGBD est centralisé ou réparti. En effet, dans un environnement réparti, il peut arriver qu'un ordonnancement global S soit décomposé en plusieurs sous-ordonnancements S_j sur différents sites j. Dans ce cas, le fait que les S_j soient sérialisables n'entraîne pas nécessairement que S est sérialisable.

Exemple 2.7 : Considérons deux transactions T_1 et T_2 et une donnée x qui est répliquée sur deux sites :

T_1 :	$R_1(x)$	T_2 :	$R_2(x)$
	$x \leftarrow x + 5$		$x \leftarrow x * 10$
	$W_1(x)$		$W_2(x)$
	$Commit_1$		$Commit_2$

Les deux transactions doivent être exécutées sur les deux sites. Considérons deux ordonnancements locaux possibles S_1 et S_2 sur chacun des deux sites :

$$S_1 = R_1(x)\ W_1(x)\ Commit_1\ R_2(x)\ W_2(x)\ Commit_2$$

$$S_2 = R_2(x)\ W_2(x)\ Commit_2\ R_1(x)\ W_1(x)\ Commit_1$$

S_1 et S_2 sont localement sérialisables puisqu'ils sont sériels, mais ne sont pas équivalents. Supposons que la valeur initiale de x soit 1. À la fin de l'exécution, x a la valeur 60 au site 1 et 15 au site 2, violant ainsi la cohérence mutuelle des deux bases de données locales.

Théorème de la sérialisabilité

Soit S un ordonnancement sur $T = \{T_1, T_2, ..., T_n\}$. Le graphe de précédence de S, noté G(S), est le graphe orienté dont les nœuds représentent les T_i et les arcs les $T_i \rightarrow T_j$ tel que, pour une donnée x, on a :

1- $R_i[x]\ \alpha\ W_j[x]$ ou

2- $W_i[x]\ \alpha\ R_j[x]$ ou

3- $W_i[x]\ \alpha\ W_j[x]$

Le graphe de précédence $G(S_0)$ de S_0 est représenté à la Figure 2.5.

Théorème : Si G(S) est acyclique alors S est sérialisable (Bernstein et Goodman, 1982).

Ce théorème peut aussi être utiliser pour vérifier qu'un Ordonnanceur est correct. Pour cela, on caractérise d'abord les ordonnancements S que l'ordonnanceur produit, puis on prouve que tous ces types d'ordonnancement ont un graphe G(S) acyclique.

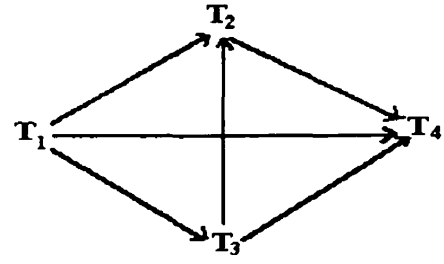


Figure 2.5 Graphe de précédence $G(S_0)$ de S_0

Remarque : Certains mécanismes de gestion d'accès concurrents font la distinction entre l'ordonnancement des conflits lecture-écriture et l'ordonnancement des conflits écriture-écriture.

2.3 Algorithmes et approches de gestion des accès concurrents

L'objectif principal de la gestion des accès concurrents est de concevoir des algorithmes qui ne produisent que des ordonnancements sérialisables, plus particulièrement en cas de conflits entre transactions. Une transaction T peut être décomposée en un ensemble d'opérations de lecture E(R) et un ensemble d'opérations d'écriture E(W). $T = E(R) \cup E(W)$.

Deux transactions T_i et T_j sont en conflit si et seulement si :

1- elles sont exécutées simultanément et

2- $E(R_i) \cap E(W_j) \neq \varnothing$ ou $E(W_i) \cap E(W_j) \neq \varnothing$.

Plusieurs versions d'algorithmes de gestion des accès concurrents ont été proposées. Cependant, en se basant sur la méthode utilisée pour réaliser la synchronisation des transactions en cas de conflits, on peut les regrouper en deux grandes classes : *pessimiste* et *optimiste*. Les algorithmes pessimistes sont ceux qui

vérifient les éventuels conflits et les résolvent avant d'exécuter les transactions; dans cette classe, on retrouve les algorithmes d'ordonnancement par verrouillage et les algorithmes d'ordonnancement par estampilles. Les algorithmes optimistes, quant à elles, laissent d'abord les transactions s'exécuter et résolvent les conflits après, s'il y a lieu; cette classe est constituée des algorithmes d'ordonnancement utilisant la certification optimiste. Cette classification est illustrée à la Figure 2.6.

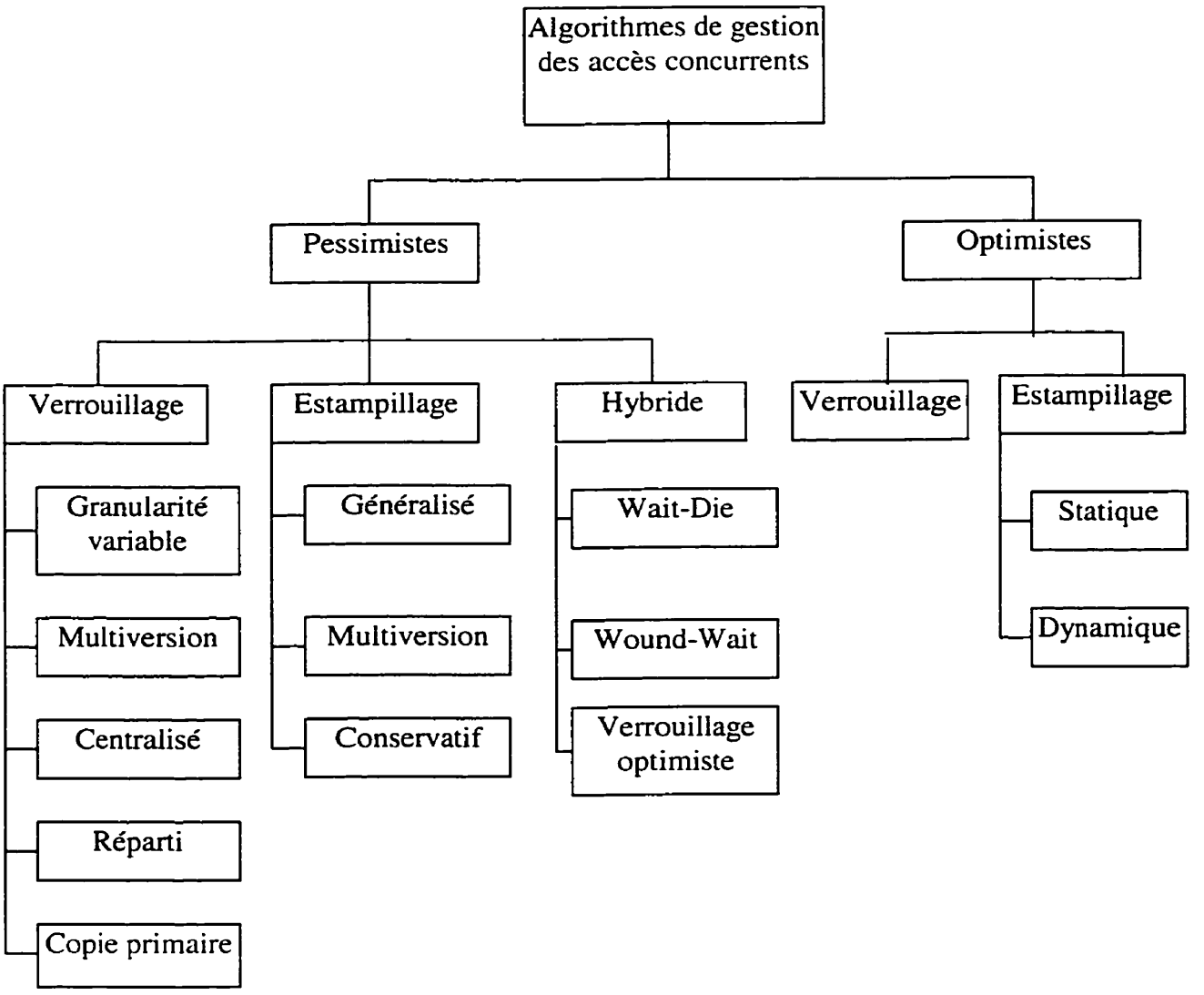


Figure 2.6 Classification des algorithmes de maintien de cohérence

Dans cette section, nous analysons les trois approches génériques que sont le verrouillage, l'estampillage et la certification optimiste. Pour chacune d'elles, nous décrivons d'abord le principe de synchronisation et l'algorithme général, ensuite nous présentons quelques versions améliorées, puis nous les comparons les unes par rapport aux autres. Nous examinons deux scénarios d'implantation possibles : le cas où l'ordonnanceur est implanté sur un seul site même si la base de donnée est répartie, et ensuite le cas ou l'ordonnanceur est réparti sur plusieurs sites. Nous étudions enfin les combinaisons qui sont possibles entre les différents algorithmes pour améliorer la performance du mécanisme d'ordonnancement.

2.3.1 Ordonnancement par verrouillage

Cette méthode résoud les conflits avant la phase d'execution en analysant l'information syntaxique sur chaque transaction. Pour l'implanter, le SGBD gère deux types de verrous sur chaque objet de la base de données : verrou de lecture et verrou d'écriture.

Verrou de lecture : une transaction verrouille une entité pour lecture dans un mode partagé. Toute autre transaction qui sollicite un verrou pour lire la même entite peut l'obtenir.

Verrou d'écriture : une transaction verrouille une entité pour écriture dans un mode exclusif. Une autre transaction ne peut plus obtenir un verrou ni en lecture ni en écriture.

Ainsi, pour accéder à une entité, une transaction doit solliciter du système le verrou approprié puis le relacher après usage. Le gestionnaire de verrous, qui est dans ce cas l'ordonnanceur, attribue un verrou à une transaction sur une entité si seulement si aucune autre transaction ne possède déjà un verrou sur la même entité dans un mode incompatible avec le type de verrou sollicité. S'il y a incompatibilité, la transaction doit attendre que tous les verrous déjà attribués sur la même entité soient libérés ou que la compatibilté soit établie, ce qui garantit un graphe de précédence sans circuit. Une analyse fine montre que les circuits sont transformés en verrous mortels. Les

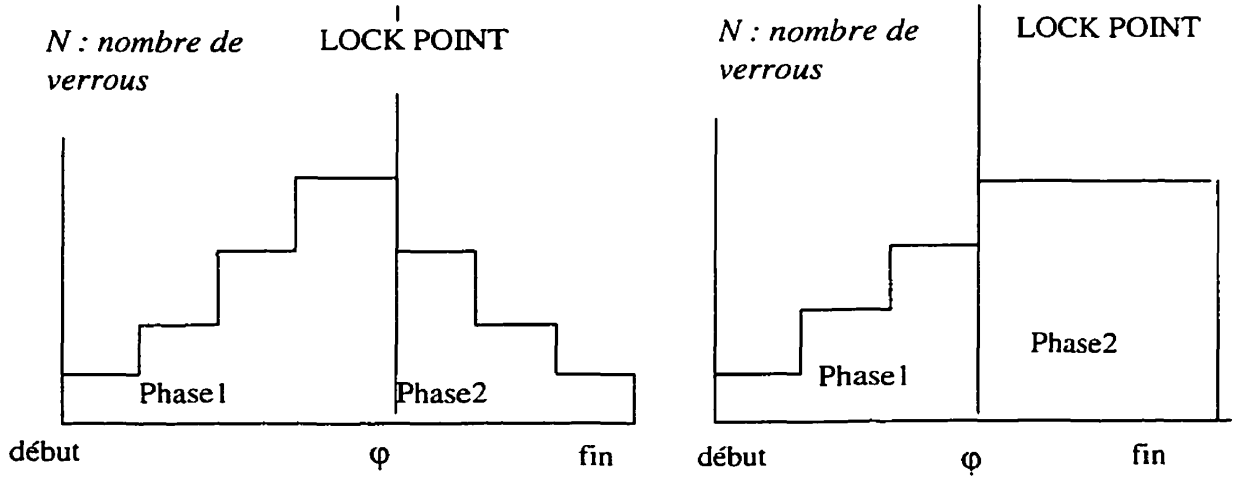
compatibilités entre opérations découlent des précédences: elles sont décrites par la matrice représentée à la Figure 2.7.

	$RL_i(x)$	$WL_i(x)$
$RL_j(x)$	compatible	incompatible
$WL_j(x)$	incompatible	incompatible

Figure 2.7 Compatibilité entre mode de verrou pour deux transactions Ti et Tj

Ordonnancement par verrouillage en deux phases

Le verrouillage en deux phases (Two Phase Locking : 2PL) est un protocole simple qui permet de garantir la correction du mécanisme. Une transaction ne peut relacher de verrous avant d'avoir obtenu tous ceux qui lui sont nécessaires. Une transaction comporte donc deux phases comme illustré à la Figure 2.8 (a) : une phase d'acquisition de verrous et une phase de relâchement.



(a) les verrous sont libérés immédiatement après usage

(b) tous les verrous sont libérés à la fin de la transaction

Figure 2.8 Graphe du comportement des transactions deux phases

Dès que la transaction libère l'un des verrou acquis, il ne lui est plus possible par

la suite d'obtenir un verrou sur une entite. Cette condition garantit un ordre identique des transactions sur les objets accédés en mode incompatible. Cet ordre est celui d'exécution des points de verrouillage maximal φ. Les Figures 2.9 et 2.10 illustrent les algorithmes généraux de verrouillage et de déverrouillage à deux phases. Plusieurs autres variantes sont resumées dans (Gardarin, 1999).

```
Bool Fonction Verrouiller (Transaction T, Granule G, Mode M) {
Cverrou : = 0;
Pour chaque transaction i ≠ T ayant verrouillé l'objet G faire {
                Cverrou : = Cverrou ∪ T.verrou(G)
        }
        si Compatible (Mode, Cverrou) alors {
                T.verrou(G) = T.verrou(G) ∪ M ;
                Verrouiller : =  VRAI ;
                }
        sinon   {
                insérer (T, Mode) dans queue de G;
                bloquer la trasaction T;
                Verrouiller : = FAUX;
                }
}
```

Figure 2.9 Algorithme général de verrouillage

```
Procédure Déverrouiller (Transaction T, Granule G) {
        T.verrou(G) : = 0;
Pour chaque transaction i dans la queue de G faire {
                si Verrouiller (i, G, M) alors {
                        enlever (i,M) de la queue de G;
                        débloquer i;
                        }
        }
}
```

Figure 2.10 Algorithme général de déverrouillage

En pratique, la sollicitation, l'acquisition et la libération du verrou sont transparentes pour la transaction et sont prises en charge par le SGBD à travers le gestionnaire de transactions. Afin de garantir l'isolation des mises à jour, les verrous sont aussi relâchés seulement en fin de transaction, lors de la validation, comme illustré à la Figure 2.8 (b). Les verrous sont demandés au moyen de l'opération *Verrouiller(T, G, M)* et relachés au moyen de l'opération *Déverrouiller (T, G)* où G représente le granule à verrouiller/déverrouiller et M le mode de verrouillage. L'application du verrouillage strict pose quelques problèmes que les versions améliorées que nous étudierons dans les sections suivantes tentent de résoudre.

Verrouillage à granularité variable

Dans une base de données, les objets à verrouiller peuvent être des tables, des pages, des tuples (BD relationnelle) ou des objets (BD objet). Une granularité variable des verrous est souhaitable, les transactions manipulant beaucoup de tuples pouvant verrouiller au niveau table ou page, celles accédant ponctuellement à quelques tuples ayant la capacité de verrouiller au niveau tuple. Le choix d'une granule fine (par exemple le tuple) minimise bien sûr les interférences ou encore les risques de conflits. Cependant, elle maximise la complexité et le coût du verrouillage.

La technique consiste à définir un graphe acyclique d'objets emboîtés et à verrouiller à partir de la racine dans un mode d'intention jusqu'aux feuilles désirées qui sont verrouillées en mode explicite. Par exemple, une transaction désirant verrouiller un tuple en mode écriture verrouillera la table en intention d'écriture, puis la page en intention d'écriture, et enfin le tuple en mode écriture. La Figure 2.11 donne la matrice de compatibilité entre les modes lecture (L), écriture (E), intention de lecture (IL) et intention d'écriture (IE).

	L	E	IL	IE
L	V	F	V	F
E	F	F	F	F
IL	V	F	V	V
IE	F	F	V	V

Figure 2.11 Compatibilité entre les modes normaux et d'intention

Verrouillage multi-versions

Le verrouillage multi-versions suppose l'existence d'au moins une version précédente d'un objet en cours de modification dans le journal géré en mémoire. Seules les transactions n'effectuant que des lectures peuvent utiliser ce mécanisme. Lors d'une lecture, si le granule est occupé par une transaction dans un mode écriture, la version antécédente du granule est livrée à la transaction qui fait la demande. Cependant, la sérialisabilité n'est pas garantie si la transaction qui accède à l'ancienne version effectue aussi des opérations d'écriture.

Le problème de blocage permanent

Ce problème survient dès qu'un groupe de transactions se coalisent, en effectuant des opérations compatibles entre elles (par exemple des lectures), contre une transaction individuelle qui désire effectuer une opération incompatible avec les précédentes (par exemple une écriture). La transaction individuelle peut alors attendre indéfiniment. Les solutions à ce problème consistent en général à mettre en file d'attente les demandes de verrouillage dans leur ordre d'arrivée et n'accepter une requête de verrouillage que si elle est compatible avec les verrouillages en cours et ceux demandés par les requêtes les plus prioritaires en attente.

Le problème du verrou mortel (deadlock)

C'est une situation dans laquelle un groupe de transactions est bloqué : chaque transaction du groupe attendant qu'une autre transaction du groupe relâche un verrou

pour pouvoir continuer. La Figure 2.12 donne un exemple de verrou mortel.

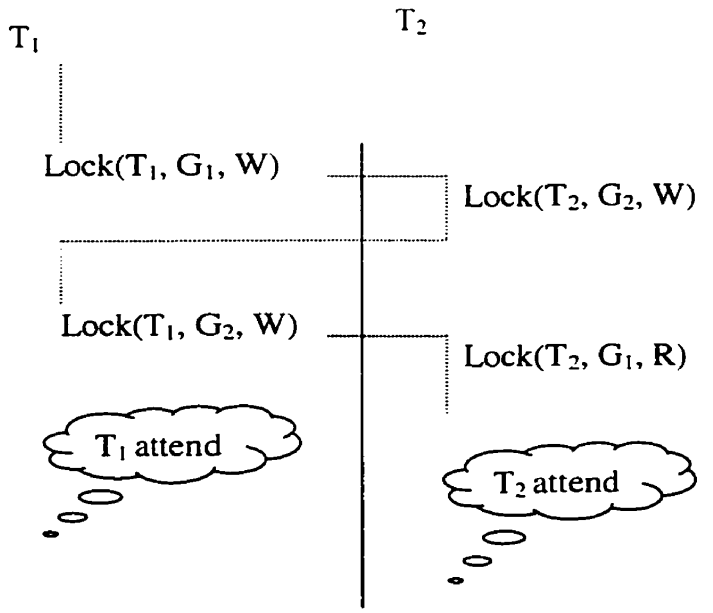


Figure 2.12 Exemple de verrou mortel

Deux classes de solutions sont possibles dans les SGBD afin de résoudre le problème de verrou mortel : la *prevention* qui empêche les situations de verrous mortels de survenir, et la *détection* qui est une solution curative consistant à supprimer les verrous mortels par reprise de transactions. Une solution simple pour prévenir le verrou mortel consiste à utiliser les delais de garde (time-outs), mais elle n'est efficace que si l'on force le developpeur des transactions à respecter un intervalle approprié de delai de garde. La prévention provoque en général trop de reprises; la détection laisse le problème se produire, détecte les circuits d'attente et les brise par annulation de certaines transactions. Un algorithme de détection d'interblocages peut se déduire d'un algorithme de détection de circuits appliqué au graphe des attentes ou des allocations. Nous présentons à la Figure 2.13 un algorithme de détection de verrou mortel; il consiste à tester si un graphe est sans circuit par élimination successive des sommets pendants

```
Bool Procédure Détecter {
T = { liste des transactions telles que N(k) ≠ 0}
G = { liste des granules allouées aux transactions dans T }
Pour chaque entrée g de G faire
        Pour chaque demande non marquée M et Tk en attente de g faire {
                Si SLOCK(k, g, Q) = VRAI alors {
                        Marquer Q;
                        N(k) = N(k) – 1;
                        Si N(k) = 0 alors {
                                Eliminer Tk de T;
                                Ajouter les granules verrouillés par Tk à G;
                                }
                        }
                }
        Si T = ∅ alors DETECTER = FAUX
        Sinon DETECTER = VRAI;
}
```

Figure 2.13 Algorithme de détection de verrou mortel

Degré d'isolation en SQL2

Le verrouillage, tel que présenté jusqu'ici, est très limitatif du point de vue des exécutions simultanées possibles. Selon une approche plus permissive laissant s'exécuter simultanément des transactions présentant des dangers limités d'incohérence, le groupe de normalisation SQL2 a défini des degrés d'isolation emboîtés, qui relaxent progressivement certaines contraines. Le groupe distingue les verrous courts relachés après exécution de l'opération, et les verrous longs relachés en fin de transaction. Le degré de verrouillage souhaité est choisi par le développeur de la transaction parmi les options suivantes :

1- Le degré 0 garantit les non pertes des mises à jour; il correspond à la pose de verrous courts exclusifs lors des écritures.

2- Le degré 1 garantit la cohérence des mises à jour; il génère la pose de verrous longs exclusifs en écriture par le système.

3- Le degré 2 garantit la cohérence des lectures individuelles; il ajoute la pose de verrous courts partagés en lecture à ceux du dégré 1.

4- Le dégré 3 atteste de la reproductibilité des lectures; il complète le niveau 2 avec

la pose de verrous longs partagés en lecture.

Un choix autre que le degré 3 doit être effectué avec précaution dans les transactions de mises à jour, car il implique des risques d'incohérence. En effet, seul le degré 3 assure la sérialisabilité des transactions.

2.3.2 Ordonnancement par verrouillage dans les BDs réparties

Dans les SGBDR, la classe des algorithmes utilisant le verrouillage comme méthode de synchronisation peut encore être subdivisée en plusieurs sous-classes, selon que la localisation du mécanisme d'ordonnancement des transactions est centralisé ou réparti. On distingue le verrouillage centralisé, le verrouillage réparti, et le verrouillage avec copie primaire (Özsu et Valduriez, 1999).

Verrouillage centralisé

La gestion des verrous sur les données de la BDR est attribuée à un seul site. Les GTs (gestionnaire de transactions) communiquent avec l'ordonnanceur (OR) du site délégué pour obtenir un verrou sur une entité de la base. Le diagramme des échanges entre les différents sites est représenté à la Figure 2.14.

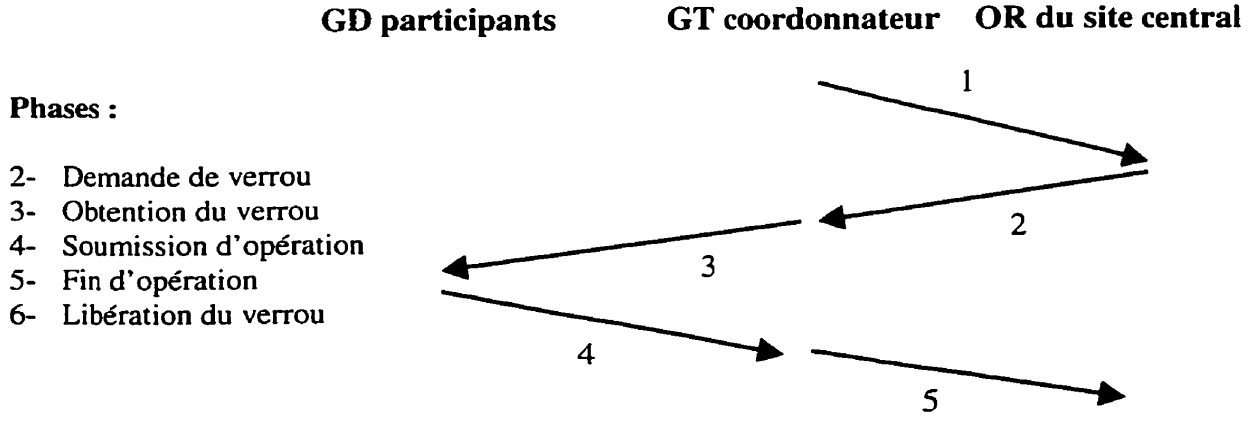


Figure 2.14 Diagramme des échanges du 2PL centralisé

La communication se fait entre le GT du site qui a reçu la transaction (désigné

comme coordonnateur). l'OR central et les GDs (gestionnaire de données) des sites participants qui abritent les données sollicitées. La principale différence entre l'algorithme de verrouillage centralisé et celui de la Figure 2.9 est que c'est le GT qui est chargé d'envoyer les opérations aux GDs et non l'ordonnanceur. L'avantage du verrouillage centralisé est que son implantation est simple par rapport au verrouillage réparti. Cependant, la centralisation du verrouillage rend le système vulnérable si le site délégué tombe en panne. De plus, la performance du système se dégrade assez vite lorsque la charge atteint un certain seuil.

Verrouillage réparti

Chaque site est responsable de la gestion des verrous sur les données qu'elle abrite. Le scénario de communication entre les sites impliqués dans l'exécution d'une transaction avec verrouillage distribué est illustré à la Figure 2.15. On note deux modifications majeures par rapport au diagramme de la Figure 2.14 : les messages sont envoyés individuellement aux gestionnaires de verrous sur chacun des sites où se trouvent les données. Ensuite, les opérations ne sont plus passées aux GDs par le GT coordonnateur, mais par le gestionnaire de verrou du site participant. L'algorithme utilisé pour implanter l'ordonnanceur réparti est similaire à celui de la Figure 2.9.

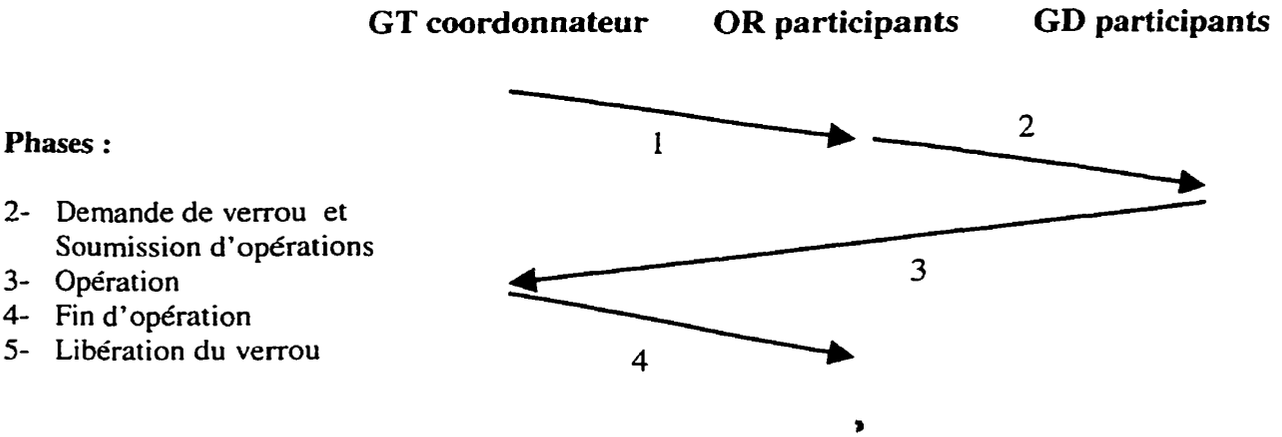


Figure 2.15 Diagramme des échanges du 2PL réparti

Verrouillage avec copie primaire

La gestion des verrous est attribuée à quelques sites, chaque ordonnanceur délégué est responsable de la gestion des verrous pour un ensemble de données de la base. Les GTs envoient cette fois-ci leurs requêtes d'obtention de verrous aux ordonnanceurs responsables des données sollicitées. Ainsi, l'algorithme utilise une copie de chaque entité de donnée (s'il sont répliquées) comme sa copie primaire. Par exemple, si une donnée x est répliquée sur les sites 1, 2 et 3, le site 1 peut être choisi comme site primaire pour la donnée x. Les transactions qui veulent accèder à x obtiennent leur verrou au site 1. L'avantage de la copie primaire est que la charge est répartie sur plusieurs sites sans causer trop de communications pour accomplir l'ordonnancement. C'est une solution intermédiaire entre l'approche centralisée et l'approche répartie. L'algorithme général est presque identique à l'algorithme centralisé. La seule difference est que la copie primaire doit être definie au départ pour chaque donnée.

2.3.3 Ordonnancement par estampillage

Bien que le verrouillage avec prévention ou détection du verrou mortel soit la technique généralement appliquée dans les SGBD, de nombreuses autres techniques ont été proposées. En particulier, l'ordonnancement par estampillage peut être utilisé non seulement pour résoudre les verrous mortels mais plus complètement pour garantir la sérialisabilité des transactions.

Algorithme de base de l'ordonnancement par estampillage

Le gestionnaire de transactions attribue au départ à chaque transaction un numéro de séquence unique appellé estampille, $ts(T_i)$. Dans un système centralisé, l'estampille correspond par exemple au nombre de transactions déjà acceptées plus 1; et dans un système réparti, l'estampille attribuée à une transaction correspond à son horodate de lancement concaténée avec l'identificateur unique du processeur qui a reçu

la transaction. Dans ce dernier cas, les estampilles de deux transactions arrivées en même temps diffèrent par le numéro du processeur en poids faible (Gardarin, 1999). Les estampilles sont ordonnées entre eux et en cas de conflit, le SGBD exécute les transactions dans un ordre specifié.

> **Règle** : Soient deux opérations en conflits O_{ij} et O_{kl} appartenant respectivement aux transactions T_i et T_k, O_{ij} est executée avant O_{kl} si et seulement si $ts(T_i) < ts(T_k)$. Dans ce cas, T_i est désignée comme la plus ancienne et T_k la plus récente. L'algorithme général d'ordonnancement par estampillage sert les opérations selon la règle premier-arrivé premier-servi.

Une façon d'implanter cet ordonnanceur consiste à conserver pour chaque objet accédé (tuple ou page), l'estampille du dernier écrivain W et celle du plus jeune lecteur R. Le contrôleur de concurrence vérifie alors que :

1- les accès en lecture s'effectuent dans l'ordre croissant des estampilles de transactions par rapport aux opérations créant une précédence, donc par rapport à l'écrivain W;

2- les accès en écriture s'effectuent dans l'ordre croissant des estampilles de transactions par rapport aux opérations créant une précédence, donc l'écrivain W et le lecteur R.

On aboutit ainsi à un contrôle très simple d'ordonnancement des accès conformément à l'ordre de lancement des transactions. En cas de désordre, il suffit de reprendre la transaction ayant créé le désordre. L'algorithme général de l'ordonnanceur par estampillage (Bernstein et Goodman, 1982) est donné à la Figure 2.16.

```
// Lorsqu'il recoit une opération ri[x] de la transaction Ti.
Fonction Lire(Ti,x)
        si      ts(Ti) < au plus grand ts de tous les wi[x] déjà acceptées alors
                annuler ri[x]
        sinon   accepter ri[x] et le traiter dès que tous les wi[x] déjà acceptées ont été
                acquitées par le GD.

// Lorsqu'il recoit une opération wi[x] de la transaction Ti.
Fonction Ecrire(Ti,x)
        si      ts(Ti) < au plus grand ts de tous les ri[x] et wi[x] déjà acceptées alors
                annuler wi[x]
        sinon   accepter wi[x] et le traiter dès que tous les ri[x] et wi[x] déjà acceptées ont
                été acquitées par le GD.
```

Figure 2.16 Algorithme général d'ordonnancement des accès par estampillage

Dans l'algorithme général d'estampillage, il y a pas d'attente, celle-ci étant remplacée par des reprises de transactions en cas d'accès ne respectant pas l'ordre. Ceci conduit en général à beaucoup de reprises.

Ordonnancement par estampillage conservatif

Contrairement à l'algorithme d'estampillage général, l'algorithme conservatif n'annule pas les opérations mais les retarde plutôt. L'ordonnanceur retient une opération jusqu'au moment où son exécution ne va pas entraîner des annulations d'autres opérations par la suite. Cette approche exige que chaque ordonnanceur reçoive les opérations de lecture ou d'écriture dans l'ordre des estampilles. L'ordonnanceur attend d'avoir dans sa queue une opération de chaque GT, puis procède aux opérations en commençant par celle qui a la plus petite estampille. Une opération est soumise au GD dès que tous ses antécédents en conflit ont été déjà acquités.

Ordonnancement par estampillage multi-versions

Comme pour le verrouillage deux phases, la stratégie d'ordonnancement par estampillage peut être améliorée en gardant plusieurs versions d'un même granule. Pour chaque objet O, le système peut maintenir :

1. un ensemble d'estampilles en écriture $\{EE_i(O)\}$ avec les valeurs associées $\{O_i\}$, chacune d'elles correspondant à une version i :
2. un ensemble d'estampilles en lecture $\{EL_i(O)\}$

Il est alors possible d'assurer l'ordonnancement des lectures par rapport aux écritures sans jamais reprendre une transaction lisant. Pour cela, il suffit de délivrer à une transaction T_i demandant à lire l'objet O la version ayant une estampille en écriture immédiatement inférieure à i. Ainsi, T_i précèdera toutes les créations d'estampilles supérieures écrivant l'objet considéré et suivra celles d'estampilles inférieures. T_i sera donc correctement séquencée. Tout se passe donc comme si T_i avait demandé la lecture juste après l'écriture de la version d'estampille immédiatement inférieure. Cependant, il est parfois possible de forcer l'ordonnancement des écritures de T_i en insérant une nouvelle version créée par T_i juste après celle d'estampille immédiatement inférieure, soit O_j. Malheureusement, si une transaction T_k ($k > i$) a lu la version O_j, alors cette lecture doit aussi être reséquencée. Cela signifie qu'il faut reprendre T_k. Afin d'éviter la reprise de transactions terminées, on préfère reprendre l'écrivain T_j avec une nouvelle estampille i' supérieure à k. Les algorithmes de lecture et d'écriture avec ordonnancement multi-versions sont représentés respectivement aux figures 2.17 et 2.18

```
Fonction Lire(Ti, O) {
        j = index de la dernière version de O;
        Tant que ts(Ti) < W(O)              // chercher la version avant Ti
                faire j = j-1;
        executer_lire(Ti , O)               // lire la bonne version
}
```

Figure 2.17 Algorithme de lecture avec ordonnancement multi-versions

```
Fonction Ecrire(Ti, Oj) {
        j = index de la dernière version de O;
        Tant que ts(Ti) < W(Oj)              // chercher la version avant Ti
                faire j = j-1;
        si ts(Ti) < R(Oj) alors
                abort(Ti)                    // annuler si lecture non dans l'ordre
        sinon executer_ecrire(Ti , Oj)       // écrire en bonne place
}
```

Figure 2.18 Algorithme d'écriture avec ordonnancement multi-versions

Ordonnancement par estampillage dans les BDs réparties

L'ordonnanceur par estampilles décrit ci-dessus s'adapte bien à une BD centralisée ou à une BD répartie, car le critère d'ordonnancement des opérations est testé localement et ne nécessite pas de communication entre sites. Les accès à l'entité x sont gérés par le même ordonnanceur qui vérifie tout simplement qu'aucune opération n'a été acceptée sur la même donnée avec une estampille plus grande.

En résumé, beaucoup d'algorithmes basés sur l'estampillage ont été proposés pour contrôler les accès concurrents. Cependant, la conservation de multiples versions pour chaque donnée accédée représente un gaspillage de l'espace de stockage. Par ailleurs, les performances de ces algorithmes restent faibles car ils provoquent tous des reprises qui deviennent de plus en plus fréquentes lorsqu'il y a un plus grand nombre de conflits. La synchronisation des horloges des différents processeurs est une condition nécessaire pour le bon fonctionnement de cet ordonnanceur, ce qui n'est pas toujours garanti en pratique. Voilà sans doute pourquoi la plupart des SGBD préfèrent encore le verrouillage à deux phases.

2.3.4 Ordonnancement par certification optimiste

Le verrouillage est jugé pessimiste à cause du fait qu'il prévient aussi des conflits qui ne surviennent en général pas. L'approche optimiste suppose que les conflits

entre transactions ne sont pas fréquents, alors le SGBD laisse les transactions s'exécuter et effectue un contrôle garantissant la sérialisabilité en fin de transaction. Une transaction est divisée en quatres phases : phase de lecture, phase de traitement, phase de certification et phase d'écriture. Pendant la phase de lecture, chaque contrôleur de concurrence garde les références des objets lus ou écrits par la transaction. Pendant la phase de certification, le contrôleur vérifie l'absence de conflits avec les transactions certifiées pendant la phase de lecture. S'il y a conflit, la certification est refusée et la transaction est défaite puis reprise. La phase d'écriture permet l'enregistrement des mises à jour dans la base pour les seules transactions certifiées. Vérifier l'absence de conflits pourrait s'effectuer en testant la non-introduction de circuit dans le graphe de précédence. L'algorithme réutilise l'estampillage, mais cette fois-ci, l'estampille est attribuée seulement à la transaction et non plus à la donnée. En plus, l'estampille n'est plus associée à la transaction au début mais juste avant la phase de validation. Chaque transaction T_i est subdivisée en un certain nombre de sous-transactions T_{ij}, exécutables sur différents sites j. Le principe consiste à mémoriser les ensembles d'objets lus (Read Set RS) et écrits (Write Set WS) par une transaction. La certification de la transaction T_i consiste à tester que les $RS(T_{ij})$ n'intersectent pas avec $WS(T_k)$ et que les $WS(T_{ij})$ n'intersectent pas avec $WS(T_k)$ ou $RS(T_k)$ pour toutes les transactions T_k lancées après T_i. L'algorithme est représenté à la Figure 2.19.

```
Bool Fonction Certifier(Ti) {
Certifier = VRAI ;
Pour chaque transaction Tk concurrente tel que ts(Tk) < ts(Tij)  faire {
      si RS(Tij) ∩ WS(Tk) ≠ ∅ ou WS(Tij) ∩ RS(Tk) ≠ ∅ ou
                              WS(Tij) ∩ WS(Tk) ≠ ∅ alors
         {
                  Certifier = FAUX;
                  Abort(Tij) ;
         }
   }
}
```

Figure 2.19 Algorithme de certification optimiste

Ordonnancement dynamique par certification optimiste

En gardant le même principe de base décrit ci-dessus, la méthode de certification peut être améliorée de l'une ou l'autre des deux façons suivantes :

1. une transaction doit relire après un certain délai les objets accédés pour tester que leur dernière mise à jour ne diffère pas avec la version lue auparavant;
2. dès qu'il y a une nouvelle mise à jour d'un objet, le GD qui abrite l'objet informe immédiatement toutes les transactions non encore validées qui ont accédé à cet objet.

Ordonnancement par certification optimiste dans les BD réparties

Depuis que l'approche optimiste fut décrite (Kung et Robinson, 1981), un grand nombre d'algorithmes ont été développés pour les SGBD centralisés ainsi que quelques extensions pour les SGBDs répartis. Agrawal *et al.* (1987) utilisent le concept de multi-version pour améliorer la performance avec des transactions qui n'effectuent que des lectures. Cependant, de longues transactions pourraient entraîner trop de reprises de transactions qui se retrouvent ainsi en blocage permanent.

En résumé, la certification optimiste permet un haut degré de concurrence entre transactions et plusieurs travaux (Agrawal *et al.*, 1987; Franaszek et Robinson, 1985) ont déjà démontré que cette méthode est plus performante que le verrouillage à deux phases dans un contexte où les conflits entre transaction sont rares. Avec la méthode optimiste, on peut augmenter facilement le taux de service juste en ajoutant de nouveaux processeurs. L'avantage principal de cette approche est qu'elle maximise l'utilisation de l'information syntaxique et sémantique sur chaque transaction. Son contrôleur de concurrence mémorise les objets accédés et effectue un test d'intersection de certains ensembles de référence lors de la validation. Un second avantage de cette méthode par rapport au verrouillage, c'est que le problème de verrou mortel ne se pose plus. L'inconvénient majeur est la tendance à reprendre beaucoup de transactions

lorsque les conflits deviennent fréquents. La méthode optimiste est donc seulement valable pour les cas où les conflits sont rares.

2.3.5 Mécanismes d'ordonnancement hybrides

Pour augmenter la performance de chacune des trois méthodes d'ordonnancement étudiées précédemment, les chercheurs tentent aujourd'hui de dériver de nouveaux algorithmes d'ordonnancement dites hybrides en combinant les différentes approches : verrouillage avec estampillage, verrouillage optimiste.

Verrouillage avec estampillage

À partir de l'estampillage, deux algorithmes ont été proposés pour prévenir les verrous mortels, WAIT-DIE et WOUND-WAIT (Rosenkrantz *et al.*, 1978). Tous deux consistent à défaire plus ou moins directement des transactions dans le cas d'attente pour éviter la formation des circuits, en donnant priorité aux transactions les plus anciennes. L'algorithme WAIT-DIE consiste à annuler les transactions qui demandent des ressources tenues par des transactions plus anciennes. La transaction la plus récente est alors reprise avec la même estampille; elle finit ainsi par devenir ancienne et par passer. Il ne peut y avoir de verrou mortel car les seules attentes possibles étant celles où une transaction plus ancienne attend une transaction récente. Le contrôle des attentes imposé par l'algorithme est précisé à la Figure 2.20.

```
Procédure Attendre (Ti, Tj)
        // Ti réclame un verrou tenu par Tj
si      ts(Ti) < ts(Tj) alors Ti attend (WAIT)
sinon   Ti annulé (DIE)
```

Figure 2.20 Contrôle des attentes dans l'algorithme WAIT-DIE

L'algorithme WOUND-WAIT permet tout type d'attente mais avec préemption. Si une transaction plus ancienne attend une plus récente, la récente est blessée

(*wounded*), ce qui signifie qu'elle ne peut plus attendre: si elle réclame un verrou tenu par une autre transaction, elle est automatiquement défaite et reprise. Le contrôle des attentes imposé par l'algorithme est précisé à la Figure 2.21. L'algorithme WOUND-WAIT provoque en principe moins de reprise de transactions et sera en général préféré.

```
Procédure Attendre (Ti, Tj)
          // Ti réclame un verrou tenu par Tj
si        ts(Ti) < ts(Tj) alors Tj est blessée (WOUNDED)
sinon     Ti attend (WAIT)
```

Figure 2.21 Contrôle des attentes dans l'algorithme WOUND-WAIT

Ordonnancement par verrouillage et certification optimiste

Il existe plusieurs façons de combiner les deux approches et nous énumérons ici quelques situations possibles.

Premier principe

Au début, une transaction accède librement aux objets, c'est-à-dire les phases de lecture, de traitement et de validation sont exécutées de façon optimiste. Même si les conflits sont détectés tardivement, l'avantage de cette méthode est que la première exécution permet déjà de copier certains objets qui ne sont pas en conflits dans la mémoire tampon de la base de donnée, ce qui réduit le temps de ré-exécution. L'ordonnanceur hybride est identique à celui du verrouillage de la Figure 2.9 sauf que l'acquisition des verrous sur les objets accédés est retardée jusqu'au début de la phase d'écriture, ce qui réduit par ailleurs le délai de possession d'un verrou.

Deuxième principe

Au début, la synchronisation des transactions est réalisée par la méthode optimiste, mais un nombre maximal de reprises est fixé à partir duquel une transaction doit réclamer les verrous pour accéder aux objets dans un mode exclusif.

L'ordonnanceur hybride est un basculement entre l'ordonnanceur optimiste et le verrouillage si la condition sur le nombre maximal de reprises est vraie ou fausse.

Troisième principe

Dans une BDR où les données sont répliquées, une transaction qui veut effectuer une écriture doit faire une demande d'acquisition de verrou à chacun des sites qui abrite un répliqua. Le verrouillage à deux phases peut s'avérer inefficace car il peut arriver qu'une transaction ne parvienne jamais à obtenir un verrou sur un objet pourtant non verrouillé si un seul des sites ne répond pas. Pour palier cet inconvénient, une amélioration possible consiste à autoriser une transaction à écrire dès qu'il a obtenu plus de la moitié des verrous sur une donnée répliquée. Il n'y a donc pas risque de conflits parce que deux transactions ne peuvent pas obtenir au même instant plus de la moitié des verrous.

CHAPITRE III

NOUVEL ALGORITHME DE GESTION DES ACCÈS CONCURRENTS

Dans le chapitre 1, nous avons fixé nos objectifs de maintien de cohérence dans les BDRs pour grappes d'ordinateurs. Dans le chapitre 2, nous avons passé en revue certains algorithmes de maintien de cohérence recensés dans la littérature. Dans le présent chapitre, nous proposons un nouvel algorithme de gestion des accès concurrents dans une BDR. Après un bref rappel des principales caractéristiques des différentes classes d'algorithmes de gestion des accès concurrents étudiés au chapitre précédent, nous présentons une description détaillée de deux algorithmes sélectionnés : celui implanté dans TelORB, et le nouvel algorithme que nous proposons. Enfin, nous comparons la performance de ces différents algorithmes.

3.1 Situation actuelle

Malgré le fait que le verrouillage en deux phases (V2P) soit considéré comme une approche pessimiste, jusqu'à nos jours tous les SGBD commerciaux utilisent encore cette méthode pour la gestion des accès concurrents. Par ailleurs, depuis que la méthode optimiste (OPT) fut introduite pour la première fois par Kung et Robinson (1981), plusieurs algorithmes de gestion d'accès optimiste ont été proposés pour les BDs centralisées et pour les BDRs. Une analyse critique des deux classes d'algorithmes - pessimiste et optimiste - est présentée par Thomasian (1998).

Les nouveaux services qui sont offerts aujourd'hui à travers les réseaux de communications utilisent très souvent les BDs; leurs exigences en terme de performance nécessitent l'exécution par le SGBD d'un volume sans cesse croissant de requêtes plus ou moins complexes des usagers. Cette situation augmente la probabilité de conflits entre transactions (notons M), ce qui se traduit dans le V2P par une augmentation du

temps d'attente moyen par transaction pour obtenir un verrou sollicité, et dans l'OPT par une augmentation du taux de réinitialisation (restart) des transactions. Comme l'a déjà démontré Thomasian (1993), lorsque M croit, plusieurs transactions sont bloquées, ce qui diminue le nombre de transactions actives et conduit à une dégradation de la performance. Les méthodes optimistes permettent un haut degré de concurrence entre transactions, ils ont aussi pour avantage de ne pas engendrer le problème de verrou mortel propre au V2P et dont les solutions envisagées pour leur détection sont parfois complexes (Knapp, 1987). Il a été démontré (Agrawal *et al.*, 1987; Franaszek *et al.*, 1992) que l'OPT offre le plus souvent un meilleur degré de performance par rapport aux algorithmes V2P strict, WOUND-WAIT ou WAIT-DIE.

Cependant, un aspect déterminant dans l'OPT, c'est l'efficacité de la méthode de validation utilisée. La méthode de validation proposée par Kung et Robinson (1981) pour les BDs centralisées cause un grand nombre de réinitialisations inutiles que l'on pourrait éviter en utilisant des estampilles. Agrawal *et al.* (1987) proposent une amélioration à cette méthode de validation en utilisant le principe de multiversion. Cependant, le problème de blocage permanent reste encore non résolu, car une transaction pourrait être réinitialisée indéfiniment.

Il est difficile de satisfaire à la fois les contraintes de préservation de cohérence forte et les exigences en terme de performance des applications émergentes. Ainsi, ni le V2P et ni l'OPT ne semble être aujourd'hui le meilleur mécanisme de gestion des accès concurrents. Certains auteurs à l'instar de Graham et Shrivastava (1988), Lausen (1982) et Thomasian (1998) ont proposé des algorithmes hybrides dont le principe consiste, soit à relaxer certaines contraintes du V2P pour permettre un degré d'optimiste, soit à synchroniser les transactions d'abord de façon optimiste puis de façon pessimiste lorsqu'une condition sur la probabilité de conflit est vérifiée. Pour les systèmes où l'on recherche à la fois la préservation de cohérence et la performance, les algorithmes hybrides semblent présenter le meilleur compromis. Par exemple, pour un SGBDR utilisé dans des applications bancaires où la préservation de cohérence forte est un aspect plus important par rapport au temps de réponse, le verrouillage en deux phases

serait le meilleur modèle, tandis que pour un SGBD utilisé dans certaines applications en temps réel, la méthode optimiste pourrait être avantageuse pour autant qu'il y ait moins de conflit entre transactions.

3.2 Critères de sélection d'un algorithme de gestion des transactions

Chacune des deux classes d'algorithmes présente des avantages mais aussi des inconvénients. Ainsi, pour choisir le meilleur algorithme, il convient d'utiliser une approche hybride, puis de relaxer certaines contraintes non primordiales pour les applications qui utiliseront la BDR. Il faudrait aussi tenir compte des caractéristiques du matériel utilisé. Au niveau matériel, il faut tenir compte des caractéristiques suivantes : la topologie du réseau, la bande passante, le processeur, la mémoire et la localisation de la BD.

- *La topologie du réseau* : pour une BDR, un algorithme centralisé serait plus performant sur un réseau en étoile, alors qu'un algorithme distribué serait préféré pour un réseau en bus ou en anneau.
- *La bande passante* : le nombre de messages échangés sur une liaison entre deux nœuds diffère d'un algorithme à un autre, ainsi le taux de transactions actifs pourrait être limité par la bande passante.
- *Le processeur* : si les processeurs utilisés dans le réseau ne sont pas rapides, il serait mieux de choisir un algorithme réparti afin de distribuer le traitement entre les nœuds, ou encore un algorithme optimiste pour éliminer les tâches de gestion du verrou et de détection du verrou mortel qu'il faut absolument traiter dans le cas du verrouillage.
- *La mémoire* : les algorithmes optimistes laissent les transactions s'exécuter simultanément même s'ils sont en conflit, mais ils nécessitent assez d'espace mémoire pour sauvegarder toutes les mises à jours qu'elles ont effectuées afin de procéder à leur validation.
- *La localisation de la BD* : le verrouillage est jugé pessimiste parce que les

transactions doivent d'abord attendre un délai pour obtenir les verrous avant d'être exécutées. Les résultats de simulation démontrent que cette critique n'est fondée que lorsque la BD réside sur disque dur Anthony *et al.* (1997); si les données résident en mémoire volatile, l'exécution des opérations est plutôt rapide et le délai moyen d'attente pour obtenir un verrou devient négligeable.

Pour satisfaire aux exigences de certaines applications, les aspects suivants doivent aussi être pris en compte lors de la conception d'un algorithme de gestion des accès : la distribution et la réplication des données, le taux d'arrivée des transactions, les contraintes de temps réel, et la probabilité de conflit.

- *La distribution des données* : un algorithme centralisé serait plus performant dans un SGBD centralisé, alors qu'un algorithme distribué serait préféré dans un SGBD réparti.
- *La réplication des données* : avec le V2P, si les données sont répliquées, une transaction ne peut effectuer une écriture sur une entité donnée que lorsqu'elle a obtenu un verrou sur chacun de ses réplicas, ce qui prolongerait davantage le temps d'attente d'obtention de verrou.
- *Le taux d'arrivée des transactions* : les expériences réalisées par Franaszek (1992) indiquent que la méthode optimiste offre un bon niveau de performance lorsque le taux d'arrivée des transactions devient élevé. En effet, on peut améliorer le taux de service des transactions simplement en ajoutant de nouveaux processeurs dans le système.
- *Les contraintes de temps réel* : certaines applications utilisant la BDR doivent s'exécuter dans un délai précis, sinon il faut les réinitialiser. Pour cela, le SGBDR doit gérer les transactions suivant un ordre de priorité.
- *La probabilité de conflit* : toutes les expériences sur les algorithmes de contrôle des accès concurrents s'accordent sur le fait que la méthode optimiste est plus performante lorsque les conflits sont moins fréquents, alors que dans le cas contraire, c'est le V2P qui est plus performant. Ainsi, il s'avère nécessaire d'étudier

au préalable le type de trafic générer par les applications pour déterminer la probabilité de conflit entre transactions. Si la valeur de la probabilité est faible, on choisit une approche optimiste; si elle est plutôt élevée, on choisit le verrouillage.

3.3 Description d'une grappe d'ordinateurs : exemple de TelORB

L'algorithme que nous développons sera implanté dans un SGBDR qui est installé sur certaines composantes matérielles du réseau téléphonique, à savoir les grappes d'ordinateurs pour la gestion des BDs des usagers et des données du trafic. Comme illustrée à la Figure 3.1, une grappe d'ordinateurs présente les caractéristiques suivantes :

- la BD est répartie et répliquée sur un ensemble de processeurs P1, P2, ..., Pn;
- la BDR est implantée en mémoire volatile;
- les nœuds qui constituent la BDR sont reliés par un réseau en bus ou en anneau;
- la bande passante est faible au niveau de l'interface SS7 qui relie le nœud d'entrée (front-end) au réseau téléphonique public à travers lequel proviennent les requêtes des usagers. Ainsi, le taux maximum des transactions entrantes dans le système est de 4000 transactions/secondes;
- les processeurs utilisés ont une vitesse moyenne de 200-300 MHz. En effet, l'un des objectifs recherchés dans une grappe d'ordinateurs, c'est de réaliser le meilleur rapport coût/performance, en utilisant des processeurs peu dispendieux.

Un SGBDR utilisé dans le réseau téléphonique gère plusieurs types de BDs, à savoir :

- HLR et VLR : ce sont des BDRs qui contiennent les données de localisation des usagers. Pour établir un appel, un processus est chargé de lire les données de localisation de l'usager appelé, tandis qu'un autre processus serait chargé de mettre à jour les données d'un usager lorsque celui-ci change de zone ou de cellule.
- BDs du trafic : Les données qu'elles contiennent sont manipulées par les applications qui gèrent les ressources du réseau ou par les opérations de maintenance, et leurs mises à jours peuvent être fréquentes.

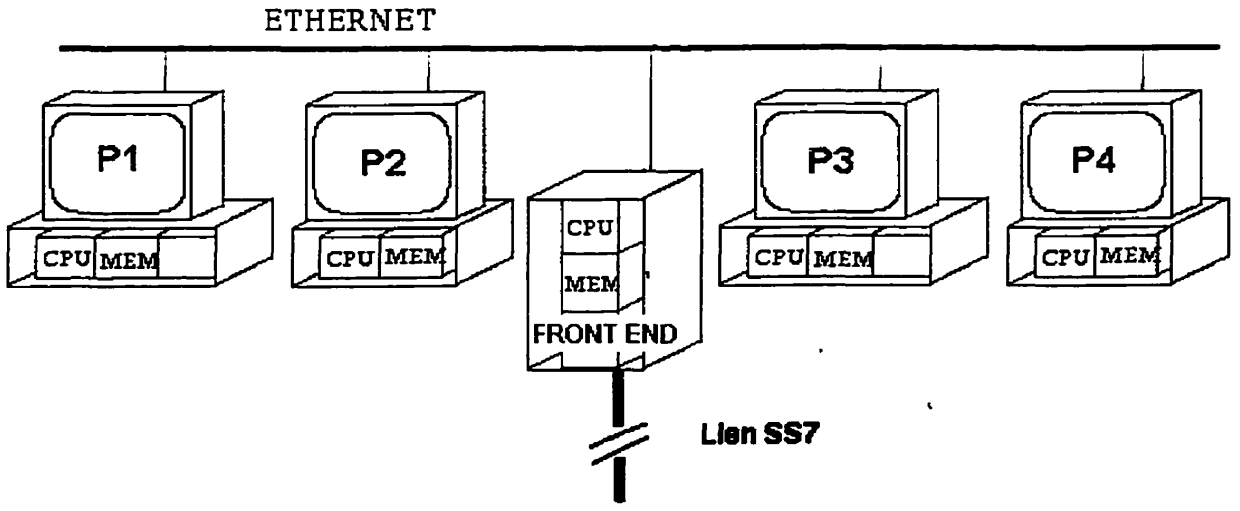


Figure 3.1 Schéma d'une grappe d'ordinateurs

La distribution du traitement sur les processeurs du réseau est faite en regroupant sur un même processeur les données d'une certaine catégorie d'usagers avec les applications qui les manipulent. La réplication est utilisée pour permettre la tolérance aux pannes et la haute disponibilité du service. En vue de satisfaire certaines contraintes de qualité de service, on définit un délai d'établissement d'une communication ou de mise à jour des données d'un abonné; les opérations de lecture ou d'écriture que ces processus entraînent au niveau de la BD sont donc sujette à des contraintes de temps réel. L'analyse du trafic réalisée par Jyhi-Kong *et al.* (1996) sur les HLRs et VLRs du réseau GSM de Taiwan montrent que le pourcentage des opérations de mise à jour des données de la BD est faible par rapport au pourcentage des opérations de lecture, ce qui suppose que la probabilité de conflits entre transactions est généralement faible. Néanmoins, cette conclusion est valable surtout pour les régions ou villes à densité égale ou inférieure à celle de Taiwan. L'écart entre les deux pourcentages peut se réduire davantage si l'on considère une région où les cellules sont de petites tailles et que la mobilité des usagers est plus importante par rapport aux données de Taiwan.

3.4 Description de quelques algorithmes sélectionnés

D'après notre analyse du problème de gestion des transactions dans une BDR et des caractéristiques des différents algorithmes déjà proposés, il ressort que ni le verrouillage, ni la méthode optimiste ne satisfait aux contraintes de maintien de cohérence et d'exigence de performance. Les algorithmes hybrides présentent une meilleure approche de solution, et l'une des plus récentes versions proposée par (Thomasian, 1998), suppose que, les sous-modules d'ordonnancement implantés à chaque nœud du SGBDR utilisent le même algorithme. Cependant, la fréquence de mise à jour des données sur la BD de trafic est différente de la fréquence de mise à jour des données sur les BDs d'usagers. Dans l'avenir, d'autres applications pourraient nécessiter de nouvelles BDs ayant chacune une fréquence de mise à jour différente. Comme hypothèse, nous supposons que l'on peut améliorer davantage la performance de l'ordonnanceur si les données sont distribuées en regroupant sur un même nœud les BDs dont les fréquences de mise à jour sont presque identiques. Comme illustré à la Figure 3.2, on pourrait alors implanter un algorithme d'ordonnancement optimiste sur les nœuds où la probabilité de conflits entre transactions est inférieure à un seuil γ, et un algorithme V2P comme mécanisme d'ordonnancement sur les nœuds où la probabilité de conflits est supérieure à γ.

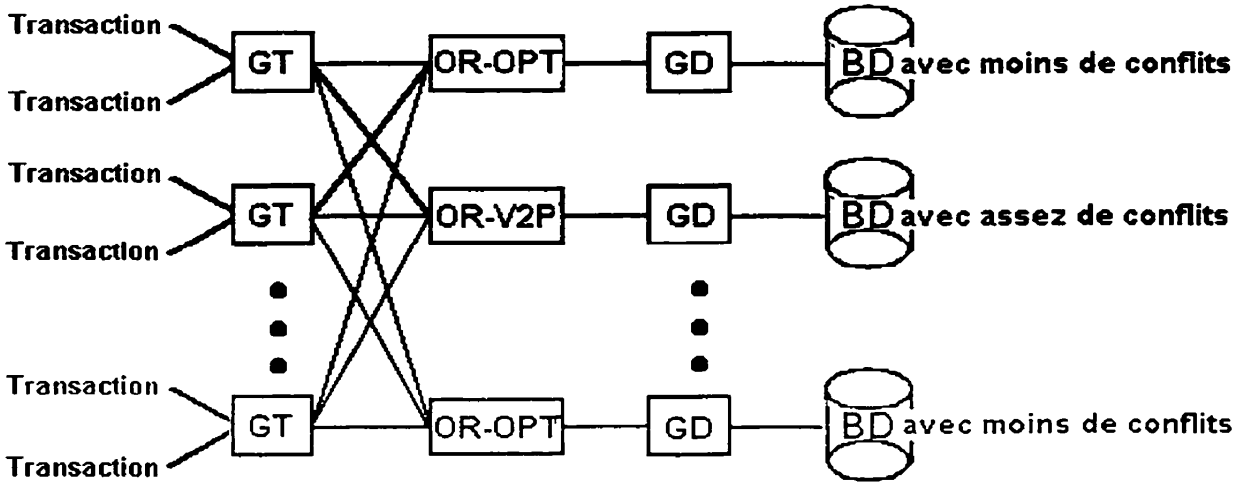


Figure 3.2 Architecture d'un SGBDR utilisant l'algorithme proposé

Une approche semblable a été adoptée dans Graham et Shrivastava (1988), mais en utilisant de vieilles versions d'algorithme de verrouillage ou d'algorithme optimiste. Par ailleurs, leur solution s'applique principalement aux SGBDRs orientés objet car les différents mécanismes d'ordonnancement sont encapsulés dans quelques objets parents et transmis aux nouveaux objets fils créés par héritage.

Pour vérifier notre hypothèse formulée précédemment, nous allons implanter deux programmes de simulation en utilisant tour à tour les algorithmes de Thomasian (1998), de Graham et Shrivastava (1988), de TelORB, et enfin l'algorithme que nous proposons. Dans la première expérience, les transactions sur la BD seront générées en simulant une application où la probabilité de conflits entre transactions serait plutôt faible et, dans la seconde expérience, nous allons considérer une application où les conflits pourraient être fréquents. Pendant les expériences, nous mesurerons certains paramètres de performance, à savoir : le temps de réponse moyen (T_m), le débit ou encore le taux de transactions servies par secondes, (T_s), le taux de réinitialisation (T_i) dans l'approche optimiste, et le délai moyen d'attente (T_v) pour obtenir le verrou dans l'approche verrouillage. Dans la suite, nous présenterons chacun des quatre algorithmes sélectionnés pour la simulation.

3.4.1 Algorithme de Thomasian

On dénombre plusieurs dizaines d'algorithmes hybrides pour la gestion des accès concurrents. Toutefois, la plupart proviennent des mêmes auteurs parmi lesquels Alexander Thomasian (1998) dont la plus récente version est présentée comme meilleur par rapport aux autres. D'après ce chercheur, son algorithme est conçu pour les BDR dont l'accent est mis sur la performance et il permet de maximiser le taux de transactions servies par secondes et de minimiser le temps de réponse. Le protocole implanté dans cet algorithme présente les caractéristiques suivantes :

1. une transaction est exécutée d'abord de façon optimiste en ignorant les verrous détenus sur les objets par d'autres transactions.
2. avant de procéder à la phase de validation globale, la transaction demande les

verrous appropriés pour chaque élément de données accédé. Ainsi, les verrous sont détenus seulement pendant la phase de validation (si celle-ci réussit), ce qui minimise le taux de conflits entre transactions par rapport à la méthode V2P strict.

3. si la validation échoue, tous les verrous déjà acquis sont retenus par la transaction pendant sa ré-exécution. Ainsi, la probabilité de réussite de la seconde phase d'exécution est élevée, si aucun nouvel objet n'est référencé; puis, cette façon permet de prévenir les réinitialisations successives et le blocage permanent. De plus, le temps d'exécution de la seconde phase est considérablement réduit car il y a moins d'accès disque.
4. le problème du verrou mortel est résolu en utilisant un paradigme de verrouillage statique, i.e., préparer les demandes de verrous pour les objets accédés par une transaction lors de sa première phase d'exécution et traiter ces demandes *dans le même ordre* à tous les nœuds.
5. la demande de verrou n'entraîne pas de messages additionnels.
6. le protocole est complètement réparti sur les différents nœuds de la BDR.

La Figure 3.3 décrit l'exécution d'une transaction T à son nœud primaire en suivant l'algorithme de Thomasian. Dans cette figure, le protocole *OCC1* signifie qu'une opération est exécutée d'abord de façon optimiste sur la dernière version valide d'un objet en ignorant les verrous détenus sur cet objet par d'autres transactions jusqu'au début de la phase de validation; le protocole *OCC2* signifie qu'une opération est ré-exécutée de façon optimiste en réclamant un verrou sur l'objet au début de la phase de lecture; *EL(T)*, l'ensemble des objets lus par T ; *EE(T)*, l'ensemble des objets écrits par T; *EL(T, S)*, l'ensemble des objets lus par T au nœud S; *EE(T, S)*, l'ensemble des objets écrits par T au nœud S; wct(O,T), numéro de version d'une copie de l'objet O tel que vu par T.

La Figure 3.4 décrit la première phase de validation qui comprend l'acquisition du verrou et la validation de T au nœud S. Une phase de l'exécution (délimitée par << ....>>) pourrait avoir lieu pendant que d'autres transactions entrent en phase de

validation. On suppose que toute opération d'écriture inclut au début une phase de lecture.

```
{phase de lecture de la première exécution, OCC1 est le protocole par défaut }
pour chaque objet O qui doit être accédé faire
        si O appartient au nœud local alors
                si (OCC2 et (une demande verrou attend O)) alors
                        mettre la demande de verrou de T en queue;
                        attendre jusqu'à ce que les verrous en conflits soient libérés;
                sinon   exécuter l'opération sur O;
                si OCC2 alors octroyer un verrou à T;
                        sinon enregistrer O avec le numéro de version courant dans
                                EL(T) ;
                                si c'est une écriture alors
                                    ajouter O à EE(T);  {modifier la copie privée de O}
                                fin;
        sinon   lancer la procédure d'accès à un objet distant;
                {attendre le système distant si les ensembles de verrous entre en conflits}
        fin;

{1ère phase de validation}
        diffuser la demande de validation aux systèmes distants contactés pendant la
        phase de lecture de T;
        obtention des verrou locaux et validation de T;
        recevoir le résultats de la validation des systèmes distants;

{2ème phase de validation }
si toutes les validations réussissent alors
        enregistrer le "commit";
        diffuser COMMIT à tous les participants;
sinon   {re-exécution}
        réinitialiser et re-exécuter la transaction;
        si aucun nouvel objet n'est référencé alors
                enregistrer le "commit";
                diffuser COMMIT à tous les participants;
        sinon
                lancer la procédure "two-phase commit" avec la validation;
        fin;
fin:
```

Figure 3.3 Exécution d'une transaction T à son nœud primaire en suivant l'algorithme de Thomasian

```
<<VALIDE := vrai ;
pour chaque O dans EL(T, S) faire
        si (un ensemble de verrou ou une demande de verrou attend O)) alors
                VALIDE := faux ;
        si conflit de verrou sur O alors
                si O dans EE(T, S) alors
                        placer la demande de verrou de E dans la queue ;
                sinon
                        placer la demande de verrou de L dans la queue ;
        sinon {pas de conflit}
                si O dans EE(T, S) alors
                        ET := T         {acquisition de verrou en écriture E}
                sinon
                        ajouter T à la liste ST   {acquisition de verrou en lecture L}
                fin;

{validation en utilisant les estampilles}
si wct(O, T) < WCT(O) alors    // numéro de version d'une copie de l'objet O
        VALIDE := faux ;
fin de pour ; >>
si VALIDE alors
        attendre que toutes les demandes de verrous à S soient obtenus ;
        mettre à jour le fichier journal ;        {précommit}
        envoyer OK
sinon
        attendre que toutes les demandes de verrous à S soient obtenus ;
        envoyer ECHEC ;
fin ;
```

Figure 3.4 Première phase de validation de T

3.4.2 Algorithme de Graham et Shrivastava

Graham et Shrivastava (1988) proposent une technique assez flexible qui permet d'adopter plusieurs variantes d'algorithmes d'ordonnancement pour implémenter un mécanisme de gestion des accès concurrents dans une BD orientée objet. Cette technique consiste à créer un ensemble de classes de base et encapsuler un algorithme

d'ordonnancement dans chacune d'elle: les nouvelles classes d'objets sont alors dérivées des classes de base et le mécanisme de gestion des accès concurrents est transmis par héritage. Certaines nouvelles classes pourraient aussi raffiner le mécanisme d'ordonnancement si nécessaire. L'article propose une définition d'une classe de base avec V2P comme premier algorithme; il suppose qu'une deuxième classe pourrait être définie et basée sur la méthode optimiste en gardant la même structure de classe. Les figures 3.5, 3.6 et 3.7 présentent respectivement la définition d'une classe de base V2P, d'une classe de base OPT et d'une classe dérivée.

```
class OrdVer : public Objet_racine
{
        enum etat {ACCEPTE, REFUSE};
        Liste_de_verrou     verrous_octroyes;
        Semaphore           *mutex;       // un autre mécanisme de contrôle d'accès
        boolean             tester_conflit_verrou(Verrou*);
public :
        OrdVer();
        ~ OrdVer();
        etat                demande_verrou(Verrou*);
        etat                libere_verrou(TransId*);
}
```

Figure 3.5 Définition d'une classe de base pour un ordonnanceur pessimiste

```
class OdrOpt : public Objet
{
        enum etat {ACCEPTE, REFUSE};
        Liste_de_operations  operations_actives;
        Semaphore            *mutex;       // un autre mécanisme de contrôle d'accès
        boolean              valider_operation(Operation*);
public :
        OdrOpt();
        ~OdrOpt();
        etat                 demande_validation(Operation*);
        void                 reinitialiser_operation(Operation*);  //
}
```

Figure 3.6 Définition d'une classe de base pour un ordonnanceur optimiste

```
class OdrVerDerive : public OrdVer
{
        ........                    // nouvel attribut ou nouvelle méthode, s'il y a lieu

public :
        OdrVerDerive();
        ~OdrVerDerive();
        .....                       // nouvelles méthodes, s'il y a lieu
}
```

Figure 3.7 Définition d'une dérivée pour un ordonnanceur pessimiste

3.4.3 Algorithme de TelORB

Il nous a paru fondamental de réaliser une simulation avec l'algorithme de TelORB pour confronter les résultats obtenus avec l'algorithme de Thomasian et celui de Graham et Shrivastava d'une part, et d'autre part avec l'algorithme que nous proposons. Nous présentons dans ce qui suit le protocole de gestion des accès concurrents implanté dans TelORB, tel qu'il nous a été décrit par l'un de ses principaux concepteurs.

DBN tient compte de deux contraintes additionnelles auxquelles l'algorithme de Thomasian ne fait pas référence : la contrainte de temps réel et la réplication des données. L'algorithme de base utilisé est le V2P, mais avec un degré d'optimisme qui se traduit par une relaxation de quelques contraintes. L'algorithme V2P strict est représenté à la Figure 3.8 et décrit à la section 2.3.1. Par rapport au V2P strict, l'algorithme de DBN relaxe certaines contraintes sur les opérations. Chaque objet de DBN comprend au moins deux réplicas. Lorsqu'une transaction veut accéder en mode écriture, elle doit verrouiller un seul réplica au début. Ce n'est qu'à partir du point de validation des mises à jour effectuées (phase de préparation du *Two Phase Commit*), que la transaction obtient les verrous pour écriture sur les autres réplicas et soumet en même temps ses modifications aux sites participants. Si cette phase échoue, la transaction est annulée

(*abort*).

Le programmeur d'une transaction doit d'abord spécifier un profil d'accès à une donnée, puis il a le choix entre deux modes de lecture : *dirtyRead* et *safeRead*. Le mode *dirtyRead*, permet d'une part à une transaction de lire la dernière mise à jour valide d'une donnée (latest commited data), même si elle est verouillée en mode écriture, et d'autre part à une autre transaction d'obtenir un verrou sollicité en écriture sur une donnée accédée en mode *dirtyRead* par une transaction en cours. On remarque aussi que l'algorithme exploite la présence de plusieurs réplicas pour implanter aussi le concept de multiversion. Le mode *safeRead* correspond quant à lui au mode de lecture du V2P strict.

```
{déclaration de variables}
msg :  Message;
dop :  Bdop; // message au Gestionnaire de Transactions
Op :   Operation;
x :    Item:  // de donnée
T :    TransactionId:
pm :   Dpmsg;  //message au Gestionnaire (processeur) de Données
res :  ValDonnee;
SOP :  EnsembleOp;
début
    repéter
      ATTENDRE(msg);
      cas où msg
        Bdop :
          Op ← dop.opm;
          x ← dop.data;
          T ← dop.tid;
          cas où Op
          Début_transaction :
                envoyer dop au processeur de données ;
          Lecture ou Ecriture
                rechercher une unité verrouillé lu tel que x ⊆ lu ;
                    si lu est déverrouillé ou le mode verrou de lu est
                     compatible avec Op alors verrouiller lu ;
                          envoyer dop au processeur de données ;
                     sinon  mettre dop dans la queue de lu ;
          Annulé ou Validé      // Abort ou Commit
                     envoyer dop au processeur de données ;
```

Suite page suivante

```
        Dpmsg :
            Op ← pm.opm;
            x ← pm.resultat ;
            T ← pm.tid ;
            si Op = Abort or Op = Commit alors
            Pour chaque unité lu verrouillé par T faire
                    libérer le verrou que possède T sur lu ;
                            si il y a plus de verrou sur lu et qu'il y a des
                            opérations en queue qui attendent pour lu alors
                    SOP ← première opération de la queue ;
                            SOP ← SOP ∪ {O | O est une opération en queue qui
                            veut verrouiller lu dans un mode compatible avec les
                            opérations dans SOP} ;
                    verrouiller lu pour chacune des opérations dans SOP;
                    pour toutes les opérations dans SOP faire
                            envoyer chaque opération au processeur ;
    jusqu'à l'infini
fin
```

Figure 3.8 Algorithme de verrouillage en deux phases strict

La relaxation de contraintes sur les lectures et sur les écritures permet de résoudre les différents problèmes de blocage entre transactions et d'accroître la performance, mais la cohérence absolue n'est plus garantie. Puisqu'il correspond à la pose de verrous courts exclusifs lors des écritures, nous pouvons dire que l'algorithme de DBN garantit la cohérence de degré 0 si l'on se réfère à la classification des degrés de

cohérence définie par le groupe de normalisation SQL que nous avons présentée à la section 2.3.1.

La Figure 3.9 illustre la déclaration d'une transaction et de son profil d'accès lors de la programmation.

```
// Créer une transaction et son profil d'accès
DicosDbAccessProfile accessProfile(
        2000,                                   // maxWaitOnLock
        100,                                    // maxNumberOfOpens
        5000,                                   // minNonInterupt
        false,                                  // commitSynchronous
        DicosDbAccessProfile::timeStampPolicy,  // dirty read access policy
        10000);                                 // timeStampThreshold

DicosDbTransaction trans1(accessProfile);
```

Figure 3.9 Déclaration d'une transaction et de son profil d'accès dans DBN

Ainsi, le développeur d'une transaction peut spécifier dans le profil un délai d'expiration (*maxWaitOnLock*) au-delà de laquelle la transaction est réinitialisée. La contrainte de temps réel conduit souvent à spécifier aussi des priorités pour limiter le taux des annulations; dans DBN, le développeur spécifie lui-même une estampille (*minNonInterupt*) lors de la déclaration du profil d'accès. Au début, une transaction a la priorité maximale tant que son délai d'exécution n'a pas encore atteint la valeur de l'estampille. Au-delà de cet instant, la transaction perd sa priorité et peut être annulée au profit d'une autre transaction en cas d'attente. En plus de satisfaire les contraintes de temps réel, ces deux valeurs de délais représentent aussi la solution préventive contre le problème de verrou mortel. Nous pouvons donc déduire des différentes caractéristiques étudiées que l'algorithme de DBN est semblable à l'algorithme WOUND-WAIT déjà présenté à la section 2.3.5, mais avec un degré d'optimisme. En effet, une transaction qui veut mettre à jour une donnée doit obtenir le verrou sur un des replicas et le reste au moment de la validation. L'algorithme intègre aussi le principe de multiversion.

3.4.4 Algorithme proposé

Nous reprenons le schéma de classe proposé par Graham et Shrivastava tel que présenté à la section 3.2.2, et nous proposons une implémentation qui utilise les deux algorithmes : TelORB (Figure 3.8) et Thomasian (Figure 3.3). Notre premier algorithme (pessimiste) contrôle les accès sur les nœuds où la probabilité de conflits entre transactions accédant aux BDs est considérable; la différence avec celui de TelORB est qu'une transaction qui veut accéder à une entité sur ces BDs doit obtenir les verrous sur tous les replicas à l'avance et non sur un seul. Notre second algorithme (optimiste) contrôle les accès sur les nœuds où soit la probabilité de conflit est faible, soit la relaxation de certaines contraintes de cohérence ne présente aucun danger; il est semblable à celui de Thomasian représenté à la Figure 3.2 où les verrous ne sont nécessaires qu'au point de validation.

Contrairement à Graham et Shrivastava (1988) qui se limite aux déclarations de classes, nous proposons une implémentation en utilisant des versions d'algorithmes récents. En plus, notre mécanisme de contrôle d'accès n'est pas encapsulé dans des objets, c'est un module de programme indépendant que l'on peut adapter aussi bien dans un SGBDR relationnel que dans un SGBDR objet. Par rapport à Thomasian, les BDs susceptibles de produire assez d'accès concurrents sont contrôlées par un second algorithme, c'est-à-dire que nous ne réutilisons pas le même algorithme sur tous les nœuds de la BDR. Dans TelORB, on ne fait pas de distinction entre les degrés de concurrence sur chaque BD : le même algorithme est utilisé sur tous les nœuds. C'est plutôt là un compromis où un seul réplica doit être verrouillé au début de la mise à jour et le reste au point de validation. Dans notre mécanisme, lorsqu'une transaction veut mettre à jour une entité de donnée, notre algorithme pessimiste réclame les verrous sur tous les réplicas au début, tandis que notre algorithme optimiste ne réclame aucun verrou jusqu'à la phase de validation.

3.5 Analyse de performance

Dans cette section, nous présentons une évaluation théorique du temps de réponse que l'on pourrait observer pour chacun des algorithmes sélectionnés. Soient :

λ le taux d'arrivée des transactions dans la BDR

N_0 le nombre total de transactions

N_c le nombre de transactions en conflits

c le taux de transactions en conflit d'après l'approche V2P; $N_c = cN_0$

ω le taux de transactions en conflit d'après l'approche optimiste; $N_c = \omega N_0$

t_e le temps moyen d'exécution des opérations d'une transaction.

Le V2P juge si une transaction est en conflit en se basant seulement sur l'information syntaxique et il prévient parfois des conflits qui n'auraient pas eu lieu, alors que l'approche optimiste résout les conflits après qu'ils aient eu lieu. Donc, moins de transactions sont supposés en conflit dans l'approche optimiste et on a toujours $\omega < c$.

Dans le cas de l'algorithme de TelORB, une transaction en conflit rejoint d'abord la queue. Soit t_q le délai moyen d'attente en queue. D'après la formule de Little, $N_c = \lambda t_q$; alors le temps de réponse moyen par transaction est donnée par :

$$t_{ORB} = (N_0 - N_c)*t_e + N_c*(t_q + t_e) / N_0$$

$$t_{ORB} = t_e - ct_e + cN_c / \lambda + ct_e = cN_c / \lambda + t_e$$

$$t_{ORB} = c^2 * N_0 / \lambda + t_e$$

Dans le cas de l'algorithme de Thomasian (optimiste), une transaction est d'abord exécutée et, s'il y a détection de conflit, elle est reexécutée comme dans le V2P en passant par une queue. Dans ce cas, le temps de réponse moyen est :

$$t_{Th} = (N_0 - N_c)*t_e + N_c* t_e + N_c(t_q + t_e) / N_0$$

$$t_{Th} = \omega N_c / \lambda + t_e + \omega t_e$$

$$t_{Th} = \omega^2 * N_0 / \lambda + t_e + \omega t_e$$

Dans le cas de l'algorithme que nous proposons, une fraction des transactions (αN_0) est exécutée sur des nœuds implantant un algorithme semblable à celui de TelORB, tandis que l'autre fraction (βN_0) est exécutée sur des nœuds implantant un

algorithme similaire à celui de Thomasian. On a aussi $\alpha + \beta = 1$. Alors, le temps de réponse moyen par transaction est donnée par la relation :

$$t_{PR} = (\alpha N_0)*[\, c^2 * (\alpha N_0) / \lambda + t_e \,] + (\beta N_0)*[\omega^2 * (\beta N_0) / \lambda + t_e + \omega t_e] / (N_0)$$

$$= \alpha*[\, c^2 * (\alpha N_0) / \lambda + t_e \,] + \beta*[\omega^2 * (\beta N_0) / \lambda + t_e + \omega t_e]$$

Lorsqu'il y a moins de conflits, $c \rightarrow 0$, et $\omega \rightarrow 0$, puisque $\omega < c$ et

$t_{Th} = t_{ORB} = t_e$

$t_{PR} = \alpha*t_e + \beta*t_e = (\alpha + \beta)*t_e = t_e$

Lorsqu'il y a assez de conflits, $c \rightarrow 1$

$t_{ORB} = N_0 / \lambda + t_e$ et on distingue deux cas pour t_{Th} :

- Si au bout du compte moins de conflits que prévisibles ont eu lieu alors $\omega \rightarrow 0$. Dans ce cas il est fort probable qu'en utilisant notre mécanisme, ces transactions manipuleraient surtout des données se trouvant dans des nœuds implantant l'algorithme optimiste de Thomasian et,

  $t_{ORB} = N_0 / \lambda + t_e$ $t_{PR} = t_{Th} = t_e$

- Si au bout du compte, la plupart des conflits prévisibles ont eu lieu, alors $\omega \rightarrow 1$ et $t_{Th} = t_e + t_{ORB}$. Dans ce cas, il est fort probable qu'en utilisant notre mécanisme, ces transactions manipuleraient surtout des données se trouvant dans des nœuds implantant l'algorithme de TelORB et :

  $t_{PR} = t_{ORB} = N_0 / \lambda + t_e$ $t_{Th} = t_e + t_{ORB}$

D'après ces résultats théoriques, on peut déduire que dans toutes les situations, notre algorithme offre le meilleur résultat par rapport aux algorithmes de TelORB et Thomasian car :

- lorsque moins de conflits sont enregistrés, $t_{PR} = t_{Th} < t_{ORB}$
- lorsque plus de conflits sont enregistrés, $t_{PR} = t_{ORB} < t_{Th}$

Cependant, nous effectuerons par la suite des simulations avec les trois algorithmes pour étayer ces résultats théoriques.

CHAPITRE IV

IMPLÉMENTATION ET MISE EN ŒUVRE

Un ordonnanceur de transactions n'est pas un programme autonome. Après la phase de conception, il faut l'implémenter en l'intégrant dans un programme complet avant de procéder aux tests. Le cycle de développement d'un SGBD peut s'avérer assez long et nous éloigner de nos principaux objectifs. Un environnement de simulation permet de modéliser le comportement des autres sous-modules qui composent le SGBD, à savoir le GT, le GD, ainsi que des évènements ou des situations comme les arrivées des transactions, l'apparition des conflits, la réinitialisation et les interblocages des transactions. Notre tâche consiste à coder l'algorithme d'ordonnancement en utilisant des classes d'objets disponibles dans cet environnement pour simuler toutes les interactions possibles entre l'ordonnanceur et les autres sous-modules du SGBD. Dans ce chapitre, nous décrivons d'abord le modèle de simulation, ensuite nous présentons les différentes classes d'objets créées lors de l'implémentation; enfin, nous décrivons les objets empruntés de l'outil de simulation CSIM18 pour compléter notre programme.

4.1 Modélisation de la BDR

Pour comparer la performance des différents algorithmes de gestion des accès concurrents, nous avons modélisé notre BDR tel que représenté à la Figure 4.1. La source de trafic (S) génère des transactions dans le système à une fréquence donnée; le gestionnaire de transactions (GT) assure la coordination de l'exécution d'une transaction qui implique le nœud local ou éventuellement des nœuds distants; l'ordonnanceur (OR) implémente localement les détails d'un mécanisme de gestion des accès concurrents; le gestionnaire de données (GD) gère l'utilisation des ressources : processeurs et entrées-sorties sur le disque local; enfin l'afficheur (A) affiche les statistiques sur le temps de

réponse, le taux de service, le taux de réinitialisation. Pour chacune des composantes du SGBDR, nous avons créé dans notre programme une classe dont les attributs caractérisent l'état de la composante, alors que les méthodes caractérisent le comportement de celle-ci. Dans la section suivante, nous présentons les classes qui modélisent les objets que nous avons déclarés comme attributs ou variables dans les classes qui modélisent les différentes composantes du SGBDR.

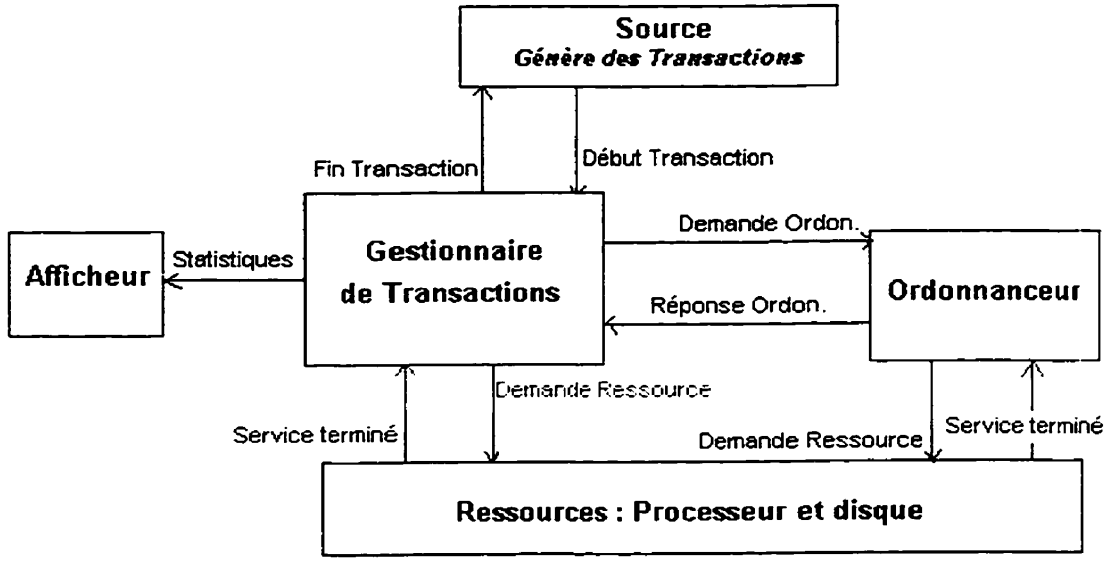


Figure 4.1 Modélisation d'une BDR

4.2 Les classes d'objets utilisés dans le programme

Cette section contient les déclarations en C++ des différents objets qui sont utilisés par les méthodes des classes qui implémentent un algorithme d'ordonnancement. Pour chaque classe, nous précisons son rôle, ses attributs et méthodes clés.

4.2.1 La classe *Verrou*

La Figure 4.2 présente la déclaration de la classe *Verrou*. Les instances de cette

classe sont créées par le GT pour l'utilisateur qui veut effectuer une opération sur un nœud utilisant un ordonnanceur par verrouillage en deux phases (OR-V2P). Ensuite, elles sont passées comme arguments à la méthode *demandeverrou()* de l'OR. La classe a pour attributs : la clé de l'objet à accéder *cle*, le mode d'accès *modeVerrou* et l'identificateur *transId* de la transaction qui l'a créée.

```
class Verrou{
        int cle;
        enum typeVerrou {LIRE, ECRIRE} modeVerrou;
        int transId;
public :
        Verrou();
        Verrou(int, typeVerrou, int);
        Verrou(Verrou &);
        int lireobjet() {return cle;}
        typeVerrou liremode() {return modeVerrou;}
        int liretransId() {return transId;}
        virtual bool operator!=(Verrou*);
  };
```

Figure 4.2 Déclaration de la classe *Verrou*

La classe fournit aussi des méthodes publiques pour vérifier l'état de ses attributs ainsi qu'un opérateur virtuel *!=* qui doit être redéclaré dans chaque classe dérivée de *Verrou*. Cet opérateur est utilisé par une méthode interne de la classe *Ordonverrou* pour tester si deux verrous entre en conflits. L'algorithme implanté par l'opérateur *!=* de la classe *Verrou* est présenté à la Figure 4.3, nous donnerons plus de détails sur son fonctionnement dans la suite.

```
bool Verrou::operator!=(Verrou* autreverrou)
{
  if ((autreverrou->liretransId() != transId) && (autreverrou->lireobjet() == cle))
```

```
{
        switch(modeVerrou
    {
        case LIRE:
                if (autreverrou->liremode() != LIRE)
                        return false;                   // il y a conflit entre les deux
modes
                break;
        case ECRIRE:
                return false;                   // il y a conflit entre les deux modes
    }
  }
  return true;                                  // il n'y a pas de conflit entre les deux
modes
}
```

Figure 4.3 Algorithme implanté par l'opérateur *!=* de la classe *Verrou*

4.2.2 La classe *Operation*

La Figure 4.4 présente la déclaration de la classe *Operation*, les instances de cette classe sont créées par le GT pour l'utilisateur qui veut effectuer une opération sur un nœud utilisant un OR-OPT.

```
class Operation{
        int cle;
        enum typeOperation {LIRE, ECRIRE} modeOperation;
        int numestampille;

public :
        Operation(int, typeOperation, int);
        Operation(Operation &);
        int lireobjet() {return cle;}
        typeOperation liremode() {return modeOperation;}
        int lireestamptrans() {return numestampille;}
        virtual bool operator!=(Operation*);
  };
```

Figure 4.4 Déclaration de la classe *Operation*

Ensuite, elles sont passées comme arguments à la méthode *enregoperation()* de l'OR. La classe a pour attributs : la clé de l'objet à accéder *cle*, le mode d'accès

modeOperation et l'identificateur *transId* de la transaction qui l'a créée. La classe fournit aussi des méthodes publiques pour vérifier l'état de ses attributs, ainsi qu'un opérateur virtuel *!=* qui est utilisé par une méthode de la classe *Ordonoptimiste* pour tester si deux opérations entrent en conflits. L'algorithme implanté par l'opérateur *!=* de la classe *Operation* est présenté à la Figure 4.5, nous donnerons plus de détail sur son fonctionnement dans la suite.

```
bool Operation::operator!=(Operation* autreoperation)
{
        if (autreoperation->lireestamptrans() < numestampille)
  {
        if (autreoperation->lireobjet() == cle)
    {
                switch(modeoperation)
        {
                case LIRE:
                        if (autreoperation->liremode() != LIRE)
                                return false;    // il y a conflit entre les deux modes
                        break;
                case ECRIRE:
                        return false; // il y a conflit entre les deux modes
        }
    }
  }
  return true; // il n'y a pas de conflit entre les deux modes
}
```

Figure 4.5 Algorithme implanté par l'opérateur *!=* de la classe *Operation*

4.2.3 La classe *Transobj*

Dans une BD, une transaction est souvent utilisée pour grouper des opérations qui doivent être exécutées de façon atomique. La Figure 4.6 présente la déclaration de la classe *Transobj*, les instances de cette classe sont créées par le GT lorsqu'un utilisateur commence une transaction, avec pour attributs l'identificateur *transId* de la transaction et l'état de la transaction. Au début, l'état prend la valeur VALIDE, le GT met à jour

l'état de la transaction après chaque opération. Le programme qui a initié la transaction doit vérifier que l'état de sa transaction est encore valide avant de passer à l'opération suivante. La classe fournit aussi des méthodes publiques pour vérifier l'état de ses attributs ainsi qu'un opérateur virtuel *!=*. Cet opérateur est utilisé par une méthode de la classe *Lchaine<Transobj>* pour comparer deux objets de type *Transobj* lors de la mise à jour de la liste des transactions actives. L'algorithme implanté par l'opérateur *!=* de la classe *Transaction* est présenté à la Figure 4.7.

```
class Transobj {
        enum etat {VALIDE, ANNULE} et;
        int transId;
        etat et;
public :
        Transobj(int,etat);
        Transobj(int);
        etat lireetat(int) {return et;}
        virtual bool operator!=(Transobj);
    };
```

Figure 4.6 Déclaration de la classe *Transobj*

```
bool Transobj::operator!=(Transobj tobj)
{
        if (tobj->lireetat() != transId)
  {
        return true;
  }
  return false;
}
```

Figure 4.7 Algorithme implanté par l'opérateur *!=* de la classe *Transobj*

4.2.4 La classe *LChaine*

Chacune des composantes du SGBDR utilise une liste, mais les objets stockés sont différents. Le GT gère une liste de transactions, l'OR-V2P gère une liste de verrous et l'OR-OPT gère une liste d'opérations. Cependant, les opérations à effectuer sur chacune de ces listes sont génériques; pour optimiser le code, nous avons alors créé une classe générique *LChaine* dont la déclaration est donnée à la Figure 4.8.

```
template <class T>

class LChaine
{
protected:
        class Noeud  {
        public :
                T Element;
                Noeud *Suivant;
        };
        Noeud *Tete;
        int Taille;
        Noeud *Courant;

public:
        bool ListVide() {return bool(Tete == NULL);};
        LChaine();
        LChaine(const LChaine& L);
        ~LChaine();
        int ListLongeur() {return Taille;}
        bool ListInsere(T classeinsere);
        bool ListRetrait(T classeretrait);
        bool ListRecherche(T & classecherche);
        void DebutNoeud();
        void ProchainNoeud();
        T NoeudCourant() {return Courant->Element;}
};
```

Figure 4.8 Déclaration de la classe *LChaine*

Cette classe est réutilisée dans chaque composante pour créer une liste chaînée

en précisant simplement le type des objets à stocker dans la liste. La classe comprend plusieurs méthodes qui sont utilisées lors de la recherche d'un élément ou lors de la mise à jour de la liste : *ListVide()*, *ListLongeur()*, *ListInsere()*, *ListRetrait()*, *ListRecherche()*, *DebutNoeud()*, *ProchainNoeud()*, *NoeudCourant()*. Dans un programme, la variable T est remplacée par le type d'objet qui est précisé lors de la déclaration de la liste. Exemple : *LChaine <Verrou>*, *LChaine <Operation>*, *LChaine <Transaction>*. Chaque élément de la liste constitue un objet de type Noeud avec un pointeur sur l'élément suivant.

4.3 Les classes modélisant les composantes du SGBDR

Cette section contient la déclaration en C++ d'une classe qui modélise les différentes composantes d'un SGBDR, et les déclarations de deux principales classes qui implémentent respectivement l'algorithme d'ordonnancement par verrouillage et l'algorithme d'ordonnancement optimiste. Pour chaque classe, nous précisons les attributs et les méthodes clés.

4.3.1 La classe *Bdr*

C'est la classe la plus globale; elle modélise l'ensemble des composantes du SGBDR, à savoir un GT, un OR et une BD. Elle a comme attributs un tableau de BD *bdrprim[]* qui représente la BDR principale et un tableau d'ordonnanceur *tabOR[]*. Lorsque la BDR est répliquée, une copie de chacun des deux tableaux précédents est affectée aux attributs *bdrRepliquee[]* et *tabORrepliq[]*. Dans ce cas, la variable *repliq* prend la valeur VRAIE. La variable *location* prend la valeur VRAIE si la BDR est en mémoire et FAUSSE sinon. Pendant la simulation, le temps de réponse considéré pour une BDR localisée en mémoire est plus faible que celui d'une BDR localisée sur disque. Pour simplifier le problème, nous supposons que le nombre de répliquas est au plus égal à 2. Le schéma *BDR* est représenté à la Figure 4.9. La classe contient aussi une liste des transactions actives *listedestrans* et une méthode privée *utiliseObjet()* qui permet de simuler les entrées/sorties sur disque. Elle fournit à la source une interface qui comprend

les méthodes suivantes : un constructeur *Bdr()* qui prend 4 arguments, le nombre de sites avec OR-V2P, le nombre de sites avec OR-OPT, la localisation de la BDR et le nombre de répliquas par entité de donnée. Pour une simulation avec l'algorithme de TelORB, tous les sites sont de type OR-V2P; pour une simulation avec l'algorithme de Thomasian, tous les sites sont de type OR-OPT. Pour une simulation avec notre algorithme, certains sites sont OR-V2P et d'autres OR-OPT; ensuite, il faut ajuster l'implémentation de certaines méthodes des classes *Ordonverrou* et *Ordonoptimiste* sans changer leur signature, afin de refléter les améliorations apportées lors la conception tel que nous l'avons décrit à la section 3.4.5.

```
class Bdr{
        int             nbSite;
        bool            location;
        bool            repliq;
        Ordonnanceur            tabOR[NUM_SITES];
        Ordonnanceur            tabORrepliq[NUM_SITES];
        facility_set            *bdrprim1;
        facility_set    *bdrRepliquee1;
        LChaine <Transobj>  listedestrans;
        void            utiliseObjet(int ,bool, int);

public :
        Bdr(int ,int ,bool ,int );
        int BDRnbSite() {return nbSite;}
        bool BDRlocation() {return location;}
        bool BDRrepliq() {return repliq;}
        void begintrans(int);
        bool lire(int, int);
        bool ecrire(int, int);
        bool commit(int);
};
```

Figure 4.9 Déclaration de la classe *Bdr*

Au début d'une transaction, la source appelle la méthode *begintrans()* pour initialiser la transaction, puis le GT lui attribue un identificateur et crée un objet

Transobj qu'il enregistre dans sa liste *listedestrans*. Ensuite, la source peut envoyer les opérations en appelant les méthodes *lire()* et *ecrire()*. Puis, le GT effectue les synchronisations nécessaires et les opérations demandées en appelant les méthodes publiques des ORs, et retourne l'état de la transaction à la source. Lorsqu'une opération est réussie, la valeur VRAIE est retournée; mais lorsqu'elle est échouée, c'est plutôt la valeur FAUSSE qui est retournée et, dans ce cas, la source doit recommencer l'opération concernée. Après la dernière opération, la source demande une validation en appelant la méthode *commit()* et, à cet instant, le GT émet un message de validation à tous les sites participants, puis il retire la transaction de sa liste et retourne l'état de cette demande à la source.

4.3.2 La classe *Ordonnanceur*

C'est une classe de base à partir de laquelle sont dérivées les classes *Ordonverrou* et *Ordonoptimiste*. Le constructeur *Ordonnanceur(char*)* prend comme argument le type d'ordonnanceur. La méthode *typeORSite()* retourne le type d'ordonnanceur, tandis que la méthode *etatSite()* retourne VRAI si le nœud est actif et FAUX si le nœud est momentanément en panne. L'implémentation de la classe *Ordonnanceur* est représentée à la Figure 4.10.

```
class Ordonnanceur {
        bool etatsite;
        char* typeOR;
public :
        Ordonnanceur();
        Ordonnanceur(char* );
        Ordonnanceur(Ordonnanceur &);
        bool etatSite() {return etatsite;}
        char *typeORSite() {return typeOR;}
};
```

Figure 4.10 Déclaration de la classe *Ordonnanceur*

4.3.3 La classe *Ordonverrou*

Le gestionnaire de verrous a pour rôle de traiter les demandes de verrous: l'OR peut octroyer immédiatement les demandes de verrous ou les placer d'abord dans une queue en cas de conflit. Notre implémentation de la classe *Ordonverrou* est représentée à la Figure 4.11. L'OR comprend une liste de verrous octroyés à des transactions actives *listeverrou* ainsi qu'une méthode privée *conflitmodeverrou()* pour comparer deux modes de verrou et vérifier si un verrou demandé entre en conflit avec un des verrous détenus par d'autres transactions. La vérification est faite en utilisant l'opérateur *!=* fourni par la classe *Verrou*. La classe fournit aussi une interface comprenant les deux méthodes *demandeverrou()* et *libereverrou()* que les GTs peuvent utiliser pour exprimer une demande ou une libération de verrou. Comme illustré à la Figure 4.12, la méthode *demandeverrou()* fait d'abord appel à *conflitmodeverrou()*. S'il n'y a pas de conflit, l'OR octroie le verrou en retournant VRAI, puis met à jour sa liste. Dans le cas contraire, il retourne FAUX et la transaction attend un certain délai pour refaire sa demande.

```
class Ordonverrou : public Ordonnanceur {
        enum resultat {OBTENU, REFUSE};
        LChaine <Verrou> listeverrou;
        bool conflitmodeverrou(Verrou*);

public :
        Ordonverrou();
        resultat demandeverrou(Verrou*);
        void libereverrou(int)
  };
```

Figure 4.11 Déclaration de la classe *Ordonverrou*

```
bool Ordonverrou::demandeverrou(Verrou* dv)
:
  bool conflit = true;
  if (conflit == conflitmodeverrou(dv))  // s'il y a conflit
        {
         cout<<"detection de conflit entre modes verrou"<<endl;
                return false;
        }
// le verrou est obtenu
         listeverrou.ListInsere(dv);
         return true;
}
```

Figure 4.12 Implémentation de la méthode *demandeverrou*

4.3.4 La classe *Ordonoptimiste*

L'OR optimiste enregistre les opérations effectuées par les transactions actives pour prendre une décision appropriée lors de la demande de validation. Notre implémentation de la classe *Ordonoptimiste* est représentée à la Figure 4.13. L'OR comprend une liste d'opérations en cours d'exécution *listeseq* qu'il met à jour soit en ajoutant une nouvelle opération lorsque celle-ci survient, soit en retirant une opération qui vient de passer le test de validation.

```
class Ordonoptimiste : public Ordonnanceur {
        LChaine <Operation> listeseq;

public :
        Ordonoptimiste();
        void enregoperation(Operation*);
        bool demandevalidation(Operation*);
  };
```

Figure 4.13 Déclaration de la classe *Ordonoptimiste*

La classe fournit une interface comprenant les deux méthodes *enregoperation()* et *demandevalidation()* que les GTs peuvent appeler pour indiquer le début d'une

nouvelle opération ou pour exprimer une demande de validation d'une opération. Si une opération passe la phase de validation, la valeur VRAIE est retournée, mais si elle échoue, la valeur FAUSSE est retournée et l'opération est recommencée. La Figure 4.14 présente l'implémentation de la méthode *demandevalidation()* que l'OR utilise pour comparer deux modes d'opération et détecter si une opération effectuée entre en conflit avec des opérations d'autres transactions. La détection utilise l'opérateur *!=* fourni par la classe *Operation*.

```
bool Ordonoptimiste::demandevalidation(Operation* op)
{
        Operation* courant;
        courant = listeseq->tete;
        while((courant != null))
        {
                if(op==courant)
                        return false;  // il y a conflit entre les deux modes
                courant = listeseq->suivant;
        }
        return true;    // l'opération est validée
}
```

Figure 4.14 Implémentation de la méthode *demandevalidation*

4.4 L'outil de simulation CSIM18

CSIM18 permet de simuler des modèles d'exécution de processus et des évènements discrets (Schwetman, 1995). C'est une bibliothèque de classes, fonctions, procédures et entêtes de fichiers que l'on peut intégrer dans un environnement de programmation comme C ou C++ pour implémenter des modèles de simulation d'applications plus ou moins complexes. Plus généralement, CSIM18 est utilisé pour modéliser une variété de systèmes : systèmes informatiques, réseaux d'ordinateurs, systèmes de communications, réseaux de transport, systèmes à microprocesseurs, etc.

Dans notre projet, nous utilisons CSIM18 pour simuler l'exécution de transactions concurrentes dans un SGBDR et des situations qui résultent d'éventuels

conflits. Dans ce type de modèle, le système est modélisé par une collection de ressources et une collection de processus qui partagent l'utilisation de ces ressources. De tels modèles sont surtout utilisés pour obtenir les estimés de performance du système réel. L'objectif de cet exercice consiste à tester différentes configurations possibles du système, différentes charges ou différentes politiques de gestion des ressources pour trouver le meilleur schéma qui permet d'atteindre le degré de performance visé. La librairie CSIM18 est disponible pour différentes plates-formes, par exemple, les ordinateurs personnels munis de systèmes d'exploitation Windows 3.1, Windows 95, Windows NT, l'OS/2 Warp ou Linux. CSIM est développé et vendu par Mesquite Software Inc, sous licence de Microelectronis and Computer Technology Corporation (MCC). Le package comprend aussi les guides d'installation et d'utilisation (Mesquite 1999) pour C et C++.

Les classes spécifiques fournies par CSIM18 comprennent entre autres :

- *Processes* : un processus est un module d'exécution indépendant ou *Thread;* les éléments actifs du système comme les clients et les serveurs sont implémentés comme des processus qui peuvent s'exécuter simultanément ;
- *Facilities* : ils sont utilisés pour modéliser les ressources actives qui sont utilisées par les processus; un objet de type *Facility* est constitué d'un ou de plusieurs serveurs et d'une queue de processus qui veulent accéder au serveur ;
- *Storages* : les unités de stockage sont utilisées pour modéliser les ressources passives qui sont allouées aux processus ;
- *Events* : les évènements sont utilisés pour synchroniser les actions de différents processus ;
- *Mailboxes* : les boîtes sont utilisées pour échanger les informations entre processus.

CSIM comprend aussi des classes pour collecter automatiquement les données de simulation à savoir :

- *Table* et *Qtable* : utilisés pour collecter les statistiques sur l'utilisation des

ressources par les processus :

- *Meter* : utilisé pour mesurer le flot d'entités qui passent par un point du système à un instant donné, et aussi pour mesurer les durées entre des passages successifs d'entités en un point ;
- *Box* : utilisé pour collecter les données sur le temps écoulé par une séquence spécifique d'activités d'un processus.

Un fichier standard de statistiques peut être généré à la fin de la simulation, pour chaque objet de type *facility, storage, table, qtable, meter, box*. Les résultats de simulation peuvent aussi être présentés sous forme d'histogrammes pour chaque objet de type *table, qtable, meter* et *box*.

En plus des classes, CSIM fournit aussi quelques outils d'assistance au développeur pendant l'implémentation de son modèle. Ces outils sont :

- l'instruction *rerun* : utilisée pour arrêter la simulation en cours et permettre au programme de reconfigurer un nouveau modèle de simulation ;
- l'instruction *reset* : utilisée au début de la simulation pour effacer les résultats de la simulation précédente ;
- *aide au débogage du modèle* : une trace peut être utilisée pour obtenir les détails sur les interactions entre processus et les différentes phases d'exécution du programme. Sur certaines plates-formes, un moniteur d'exécution permet au développeur de suivre l'exécution du programme pas à pas et de localiser les fautes éventuelles.

Dans la section suivante, nous décrivons un exemple de programme de simulation utilisant CSIM18.

4.5 Programme de simulation

Le système SGBDR que nous voulons simuler est composé d'un ensemble de nœuds comprenant chacun une BD, un GT et un OR. À chaque instant, un ensemble de transactions entre dans le système pour utiliser les données et un autre ensemble de

transactions quitte le système. Un squelette de notre programme de simulation est représenté à la Figure 4.15.

```
void sim(int argc, char *argv[])
{
    prt_intro();
    init_parms();
    while(get_parms() != QUIT) {
            if (run_ct++ > 0)
                    rerun();
            set_model_name("Simulation d'un ordonnanceur BDR");
            create("sim");
            cnt = NARS;
            for (int i=1; i <= nars; i++) {
                    hold(expntl(iatm));
                    transaction();
            }
            done.wait();
            report();
            mdlstat();
    }
    prt_exit();
}
```

Figure 4.15 Implémentation de la procédure principale

Dans CSIM, un processus est une procédure qui exécute l'instruction *create("transaction")* qui comprend deux actions principales : établir la procédure qui exécute cette instruction comme un processus autonome et retourner le contrôle au point d'appel permettant ainsi à un nouveau processus d'être créé. Notre programme comprend deux types de processus : le processus principal *"sim"* qui initialise le modèle de simulation et génère les arrivées des transactions en variant les intervalles d'inter-arrivées, et le processus *"transaction"* qui simule le comportement d'une transaction. Plusieurs transactions qui sont des instances du processus *transaction* peuvent être actives en même temps, les unes utilisant des ressources, tandis que les autres arrivent

ou attendent en queue. Chaque entité de donnée est codée comme une ressource active *facility* gérant elle-même à chaque instant une queue de transactions qui veulent utiliser la ressource. Chaque BD est codée comme un ensemble de ressources actives ayant chacune sa propre queue, *facility_set*; dans le cas où nous considérons la réplication, chaque entité de donnée existe en deux copies mais avec une seule queue; on utilise alors la classe *facility_ms*, qui permet de créer une ressource active multi-serveurs avec une seule queue. En utilisant la fonction *void facility::inf_srv()*, on a toujours une copie disponible pour la ressource *facility*, et il n'y a jamais d'attente en queue. Nous utilisons cette fonction dans le code de l'ordonnanceur optimiste, car le protocole est d'office non-bloquant et il n'y a pas de queue. Nous utilisons encore cette fonction dans le code de l'ordonnanceur V2P pour simuler les accès partagés en lecture qui sont aussi non-bloquants.

La variable *"clock"* contient le temps écoulé depuis le début de la simulation. L'instruction *"hold"* entraîne une pause du processus qui l'exécute. À chaque fois qu'un processus marque une pause, un autre processus prend la relève et c'est ainsi que le parallélisme est simulé dans CSIM18. Dans notre programme, l'instruction *"hold(expntl(iart))"* modélise le temps d'inter-arrivée entre les transactions entrantes, tandis que l'instruction *"hold(expntl(svrt))"* modélise le temps d'utilisation de la ressource. Lors de la simulation, le temps d'utilisation de la ressource lorsque la BDR est supposée en mémoire vive est le tiers du temps d'utilisation de la même ressource lorsque la BDR est supposée sur disque, afin de refléter la réalité entre le temps d'un accès mémoire par rapport à un accès disque. Lorsque nous tenons compte de la contrainte temps réel, chaque processus qui attend en queue pour accéder à une ressource *facility* doit exécuter périodiquement la fonction membre *long facility.time_reserve(long timeout)*, ensuite il quitte la queue soit parce qu'il a gagné l'accès à la ressource ou parce que le délai de garde fixé est expiré. La fonction membre *void set_priority(long new_priority)* permet d'affecter un ordre de priorité à un processus.

Autres procédures

prt_intro() : le programme principal appelle cette fonction au début pour instruire l'usager et l'informer des différentes phases de la simulation, puis il l'invite alors à entrer les paramètres de configuration du programme.

init_parms() : initialise les paramètres du programme avec les valeurs entrées par l'usager.

transaction() : contient les instructions exécutées par une transaction; pour simplifier le programme, nous avons considéré que chaque transaction comprend toujours deux opérations sur des données différentes choisies de façon aléatoires dans la BDR. Pour provoquer des conflits, nous utilisons une variable globale *probaconflit* et nous varions le pourcentage des opérations d'écriture par rapport au pourcentage des opérations de lecture.

done.wait() : attendre jusqu'à ce que la dernière transaction quitte le système.

prt_resultats() : cette fonction est appelée en fin de simulation et publie le sommaire des statistiques d'utilisation des ressources par les processus. Les données mesurées concernent : le nombre d'arrivées, le temps moyen de service, le taux de transactions servies par unité de temps, le temps d'attente moyen par transaction dans une queue, le temps de réponse moyen ou temps total passé par une transaction dans le système avant d'être complétée (prenant en compte le temps de service) et le temps d'attente en queue (dans le cas OR-V2P) ou le temps de ré-exécution (dans le cas OR-OPT), le taux de transactions réinitialisées, le facteur d'utilisation des ressources.

mdlstat() : génère les statistiques sur l'utilisation des ressources par les transactions.

4.6 Mise en œuvre du programme de simulation

Notre programme de simulation comprend trois modules exécutables indépendants, dont le premier implante l'algorithme que nous proposons, le second l'algorithme de TelORB et le troisième celui de Thomasian. Au début de la simulation, le programme informe l'usager des différentes phases qui suivront, puis l'invite à entrer

au clavier les paramètres de configuration. L'usager doit donc répondre à chaque fois par "oui" ou par "non" si l'une des contraintes suivantes s'applique : aucun conflit entre transactions, BD sur disque, réplication, temps réel. Par défaut, le programme considère que la BD est localisée en mémoire vive, que le taux de conflit est de 1/6 du nombre de transactions et qu'aucune contrainte ne s'applique. Ensuite, l'usager peut exécuter plusieurs sessions avant de quitter le programme et, au début de chaque session, il doit préciser la valeur du taux d'arrivées et le nom du fichier dans lequel seront reportés les résultats. Dans cette expérience préliminaire, nous avons considéré le programme par défaut et nous avons fixé des valeurs pour certains paramètres comme suit :

Svtmp :	Temps de service	=	1.5 sec
Iatmp :	Temps interarrivée	=	1.0 sec
Taille d'une transaction		=	2 opérations

En appliquant successivement les différentes valeurs de taux d'arrivées, le module qui implante l'algorithme que nous proposons génère une série de fichiers de résultats sous le même répertoire que le programme principal, et les différentes valeurs intermédiaires que nous avons sélectionnées sont présentées à la Figure 4.16.

Taux d'arrivées	Taux de sortie	Temps de réponse (sec)
100	2	0,304
500	10	0,359
1000	19,9	0,443
5000	20	0,558
10000	19	0,586

Figure 4.16 Taux de sortie et temps de réponse pour une BD en mémoire

Les représentations graphiques du taux de sortie et du temps de réponse sont données respectivement à la Figure 4.17 et à la Figure 4.18.

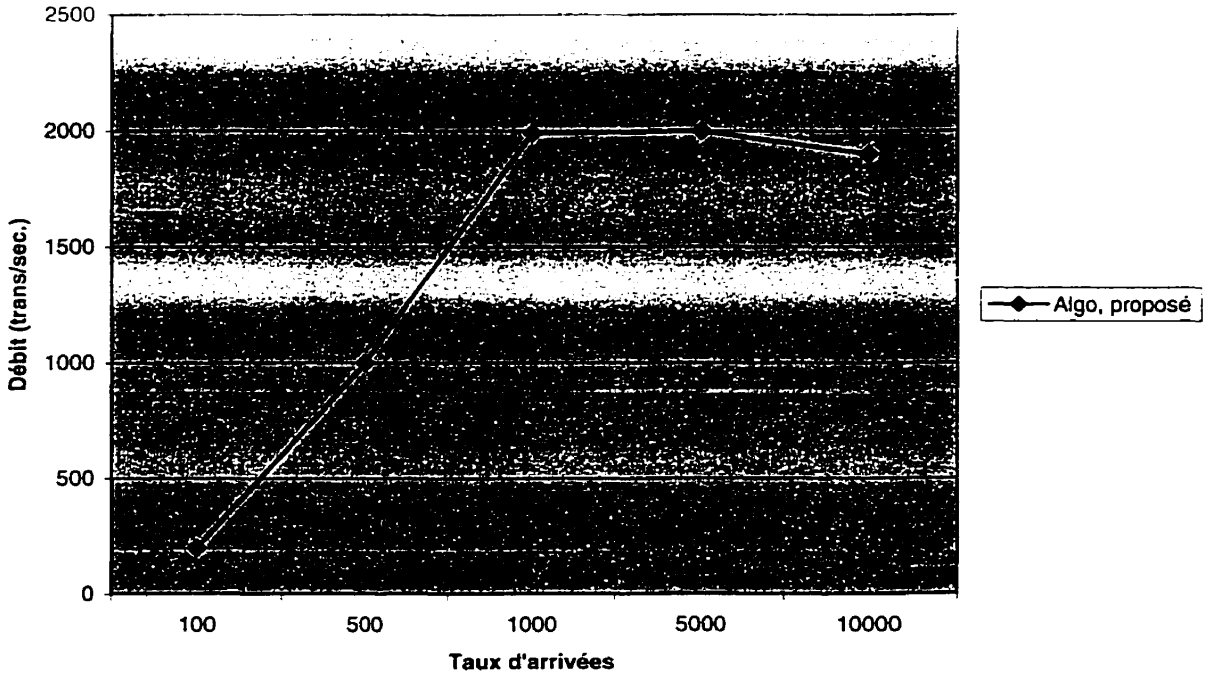


Figure 4.17 Taux de sortie en fonction du taux d'arrivée

Notons que, pour les mêmes données de simulation, les résultats obtenus après deux exécutions successives peuvent être différents, mais l'écart entre eux est le plus souvent très faible. Pour plus de précision, nous répétons chaque session trois fois, et puis nous sélectionnons la valeur intermédiaire de temps de réponse ou de taux de sortie. On observe que la courbe représentant le taux de sortie se sature à partir d'un point correspondant au seuil où le processeur a atteint la capacité de traitement maximal, et qu'elle décroît par la suite si on augmente davantage le taux d'arrivées. On observe aussi que le temps de réponse moyen croît au fur et à mesure que l'on augmente le taux d'arrivées. Ces résultats préliminaires présagent du bon comportement du module qui implante notre algorithme. Dans le chapitre suivant, nous soumettrons notre algorithme à une série de tests en appliquant différentes contraintes pour déterminer leur influence sur la performance globale du système. Puis, nous comparerons ses performances avec celles des algorithmes de TelORB et de Thomasian appliqués au même problème.

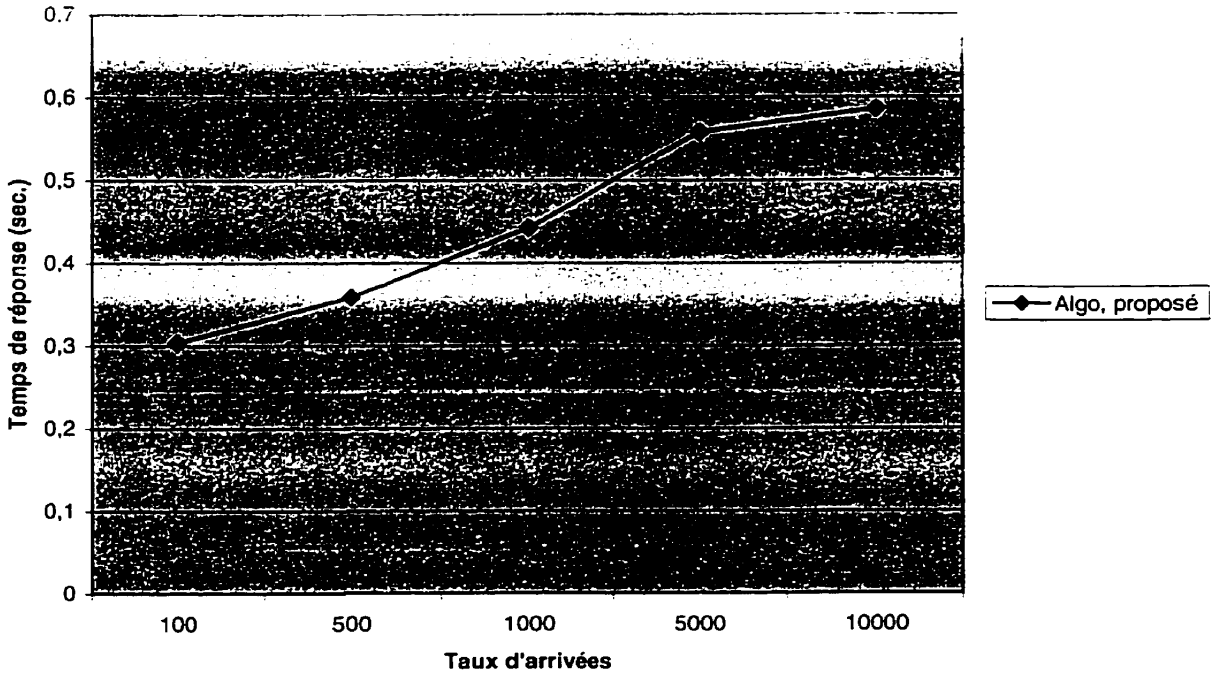


Figure 4.18 Temps de réponse en fonction du taux d'arrivée

CHAPITRE V

ANALYSE DES RESULTATS

Dans ce chapitre, nous présentons une analyse des résultats des expériences que nous avons réalisées avec CSIM et DBN pour tester les performances de notre algorithme en les comparant à chaque fois avec celles des algorithmes de TelORB et de Thomasian. Dans chaque expérience, nous varions le taux d'arrivées de transactions, et nous mesurons le débit (nombre de transactions complétées par seconde) ainsi que le temps de réponse. Nous terminons ce chapitre en proposant quelques améliorations que l'on pourrait inclure dans les prochaines versions du système TelORB.

5.1 Plan d'expériences

La performance d'un mécanisme d'ordonnancement des accès concurrents peut être mesurée à l'aide de plusieurs métriques tels que le débit, le temps de réponse, le nombre de messages échangés pour accomplir la synchronisation ou le taux d'opérations reexécutées dans le cas de l'OPT. Cependant, certains paramètres propres au système ou aux applications peuvent influencer l'indice de performance considéré : le taux d'arrivées des transactions, le nombre de répliquas par donnée, la taille des transactions, le degré de concurrence entre les transactions, le nombre de sites ou de processeurs sur lesquels la BD est répartie. À la section 3.2, nous avons énuméré les différentes contraintes des systèmes et les exigences des différentes applications dont il faut tenir compte lors de la sélection d'un algorithme performant qui garantit aussi la cohérence et nous avons constaté qu'il est généralement difficile de concilier les deux aspects. Notre plan d'expériences vise à déterminer l'algorithme qui offre le meilleur compromis cohérence-performance pour différentes contraintes possibles. Ces contraintes correspondent aux différents schémas de configuration qu'un administrateur de systèmes de bases de données recherche souvent en vue de satisfaire les exigences de

performance et de haute disponibilité des applications. Nous considérons les schémas de configuration suivants : SGBDR en mémoire vive, SGBDR sur disque, SGBDR avec réplication de données, SGBDR temps réel.

Comme dans d'autres expériences (Bhargava, 1999; Anthony *et al.*, 1997; Michael et Miron, 1988; Thomasian, 1998), nous varions le taux d'arrivées des transactions dans une plage de valeurs (100 à 10000 arrivées par seconde) qui inclut d'une part les taux d'arrivées observés sur les applications HLR et SCP d'Ericsson et, d'autre part, les valeurs considérées par d'autres chercheurs pour simuler des applications plus ou moins complexes. Nous mesurons le débit et le temps de réponse en appliquant d'abord l'algorithme de TelORB, puis celui de Thomasian et enfin celui que nous proposons. Nos simulations ont été réalisées sur un ordinateur PC ayant les configurations décrites ci-dessous. Sur le plan matériel :

- 1 processeur Pentium de 600 MHz
- 128 Mo de RAM
- un disque SCSI de 6 Go.

Sur le plan logiciel :

- le système d'exploitation Windows 98,
- l'environnement de programmation Borland C++ v.5.02
- l'outil de modélisation et de simulation CSIM18.

5.2 Comportement d'un SGBDR en mémoire vive

Dans cette simulation, nous distinguons une situation de conflits possibles entre les opérations des transactions concurrentes et une situation où les conflits seraient plutôt rares. Nous supposons également qu'il n'y a pas de réplication de données, ni de contrainte de temps réel. Nous varions le nombre d'arrivées par seconde (NARS) entre 100 et 10 000 et les paramètres fixés prennent les valeurs suivantes :

Svtmp :	temps de service pour une lecture	=	1.0 sec
	temps de service pour une écriture	=	1.5 sec

Iatmp :	temps interarrivée	=	1.0 sec
Nb_sites :	nombre de sites	=	8
Nb_obj :	nombre d'objets/sites	=	100
Taille d'une transaction		=	2 opérations

5.2.1 Cas où les conflits sont possibles

L_prob :	taux des opérations de lecture	=	5/6
E_prob :	taux des opérations d'écriture	=	1/6

Le fait d'avoir une portion (1/6) d'opérations d'écriture a pour effet de provoquer quelques conflits d'accès dans la BDR. Les résultats de simulation obtenus sont représentés aux figures 5.1 et 5.2. Nous pouvons noter dans le cas de l'algorithme de Thomasian, qu'au fur et à mesure que le taux d'arrivées augmente, le débit croît plus vite par rapport à l'algorithme de TelORB puis se stabilise entre certaines valeurs avant de décroître.

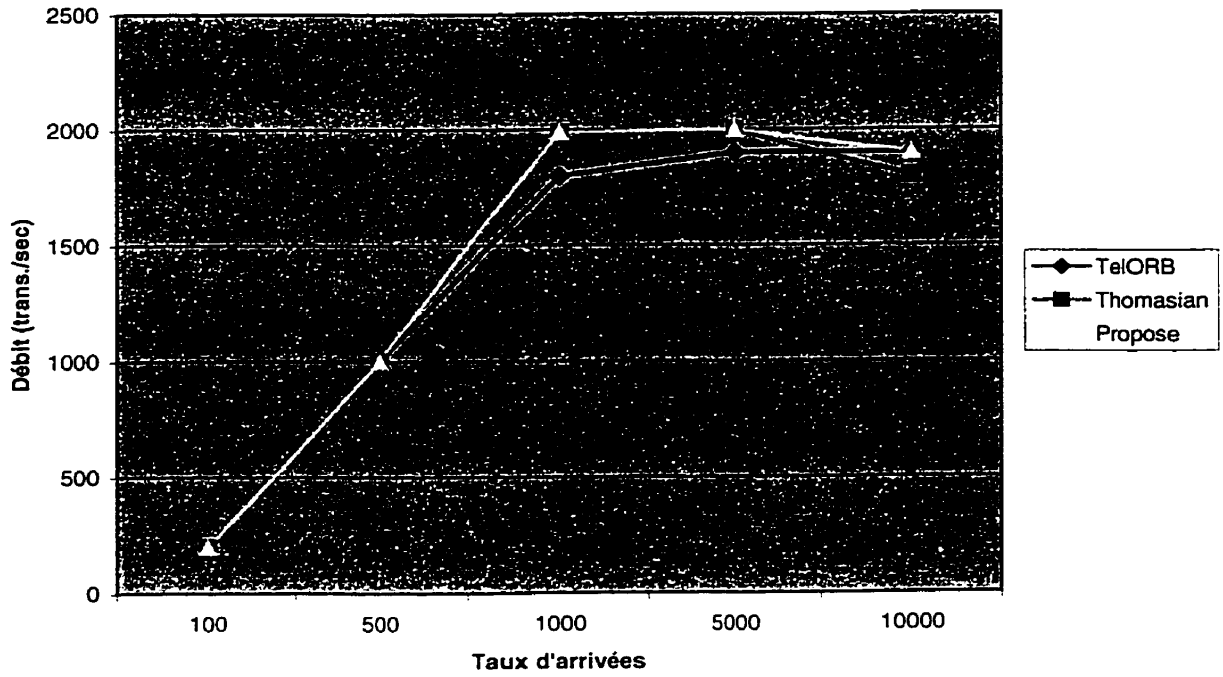


Figure 5.1 Taux de sortie en fonction du taux d'arrivée en présence de conflits

La courbe de débit avec Thomasian décroît considérablement et se retrouve finalement en dessous de celle de TelORB qui se stabilise plutôt. L'algorithme proposé quant à lui, suit au départ la même évolution que celui de Thomasian mais décroît moins vite en tendant à rejoindre la courbe de TelORB. On constate aussi que l'algorithme de TelORB fournit le meilleur temps de réponse pour chaque valeur de taux d'arrivées considérées. La courbe des valeurs observées avec l'algorithme que nous proposons se rapproche parfois vers la courbe de TelORB et parfois vers la courbe de Thomasian. Cette situation est une conséquence du fait que notre algorithme fait une sorte de commutation en utilisant la méthode de Thomasian lorsque la probabilité de conflits est négligeable et l'algorithme de TelORB lorsque la probabilité de conflits devient considérable.

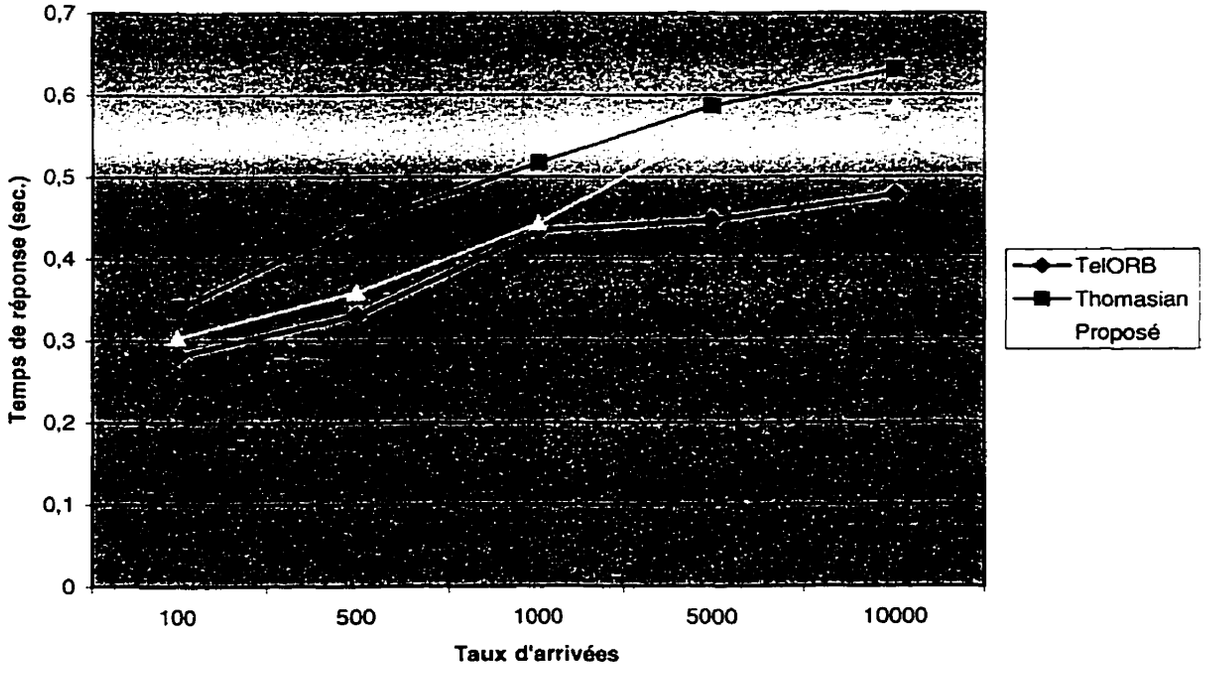


Figure 5.2 Temps de réponse en fonction du taux d'arrivée en présence de conflits

Ces résultats nous amènent à conclure que, pour des applications qui engendrent souvent des conflits entre transactions concurrentes dans la BDR, le meilleur compromis

cohérence-performance est atteint en utilisant l'algorithme de TelORB. Notons enfin qu'il faut considérer une marge d'erreur de 5 à 10 % sur chacune des valeurs obtenues lors des expériences. En effet, pour obtenir chaque point de la courbe, nous répétons la simulation trois fois et puis nous retenons la valeur qui est intermédiaire.

5.2.2 Cas où les conflits sont rares

L_prob : taux des opérations de lecture = 1

E_prob : taux des opérations d'écriture = 0

En théorie, lorsqu'il ne reçoit que des opérations de lecture, l'ordonnanceur, qu'il soit V2P ou OPT, laisse les opérations s'exécuter simultanément puisqu'il y a aucun risque de conflit, donc une opération n'est ni retardée, ni réinitialisée. Les résultats de simulation obtenus dans ce cas sont représentés aux figures 5.3 et 5.4.

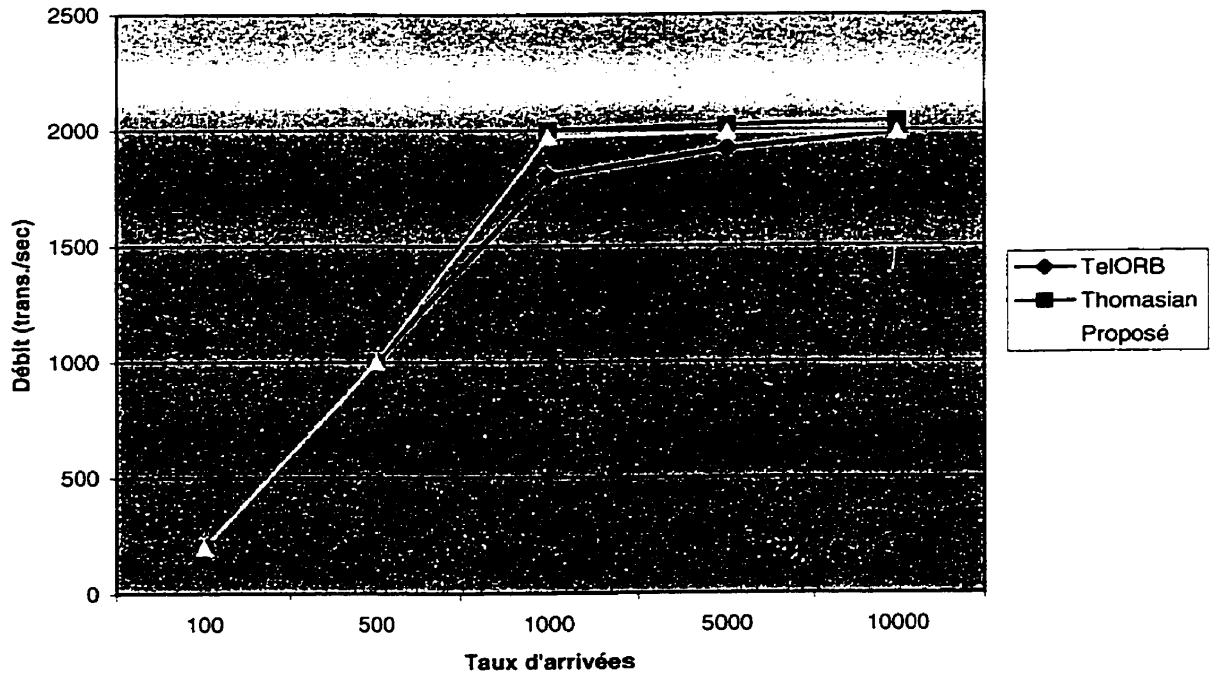


Figure 5.3 Taux de sortie en fonction du taux d'arrivée en absence de conflit

Nous pouvons noter qu'au fur et à mesure que l'on augmente le taux d'arrivée, le

débit tout comme le temps de réponse garde presque la même valeur pour les 3 algorithmes considérés avec un léger avantage de l'algorithme de Thomasian sur les deux autres. Ces résultats nous amènent à conclure que, pour les applications qui n'engendrent pas souvent des transactions en conflits dans la BDR, le meilleur compromis cohérence-performance est atteint par l'algorithme de Thomasian.

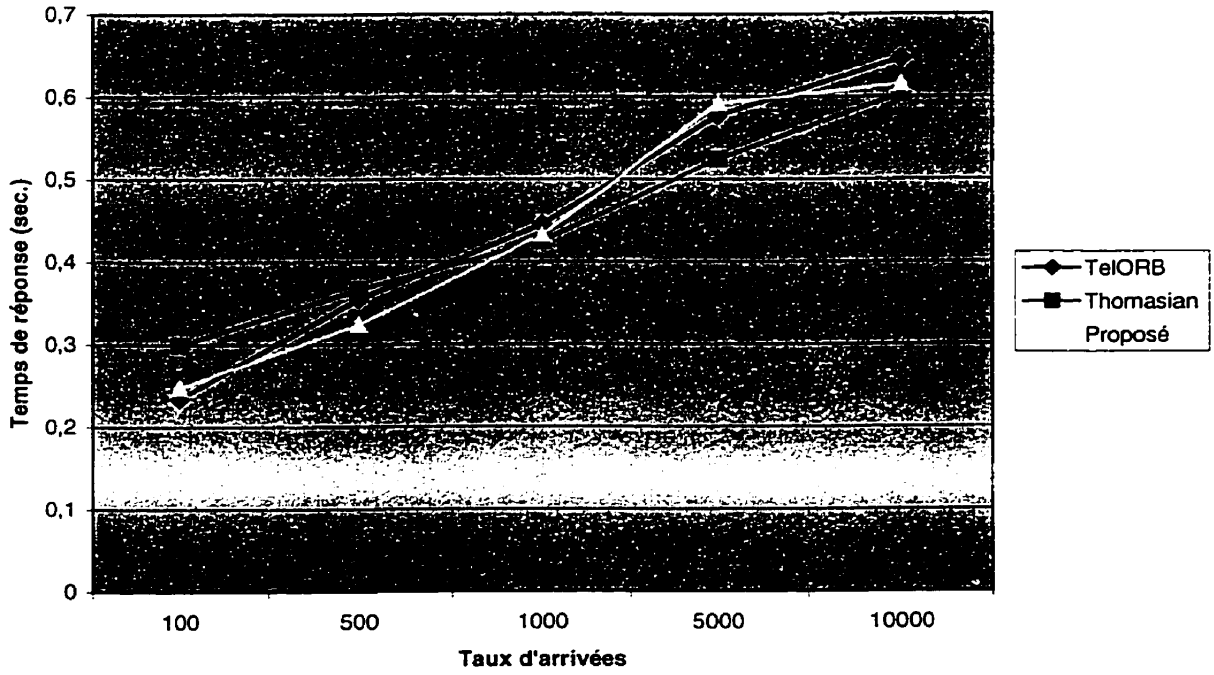


Figure 5.4 Temps de réponse en fonction du taux d'arrivée en absence de conflit

5.3 Comportement d'un SGBDR sur disque

Dans cette simulation, nous supposons une situation de conflits possibles entre les opérations des transactions concurrentes. Le temps d'accès disque est le principal paramètre à considérer pour simuler le comportement d'un SGBDR localisé sur disque dur. En pratique, le délai nécessaire pour accéder une donnée sur disque est plus important que celui qu'il faut pour accéder la même donnée si elle résidait en mémoire. Nous supposons également qu'il n'y a pas de réplication de données, ni de contrainte de temps réel. Nous varions le nombre d'arrivées par seconde (NARS) entre 100 et 10 000

et les paramètres fixés prennent les valeurs suivantes :

Svtmp :	temps de service pour une lecture	=	40.0 sec
	temps de service pour une écriture	=	60.0 sec
Iatmp :	temps interarrivée	=	30 sec
Nb_sites :	nombre de sites	=	8
Nb_obj :	nombre d'objets/sites	=	100
L_prob :	taux des opérations de lecture	=	5/6
E_prob :	taux des opérations d'écriture	=	1/6
Taille d'une transaction		=	2 opérations

Les résultats de simulation obtenus sont représentés aux figures 5.5 et 5.6. Nous pouvons noter que, pour chacun des trois algorithmes, le débit se sature très vite et demeure faible au fur et à mesure que le taux d'arrivées augmente.

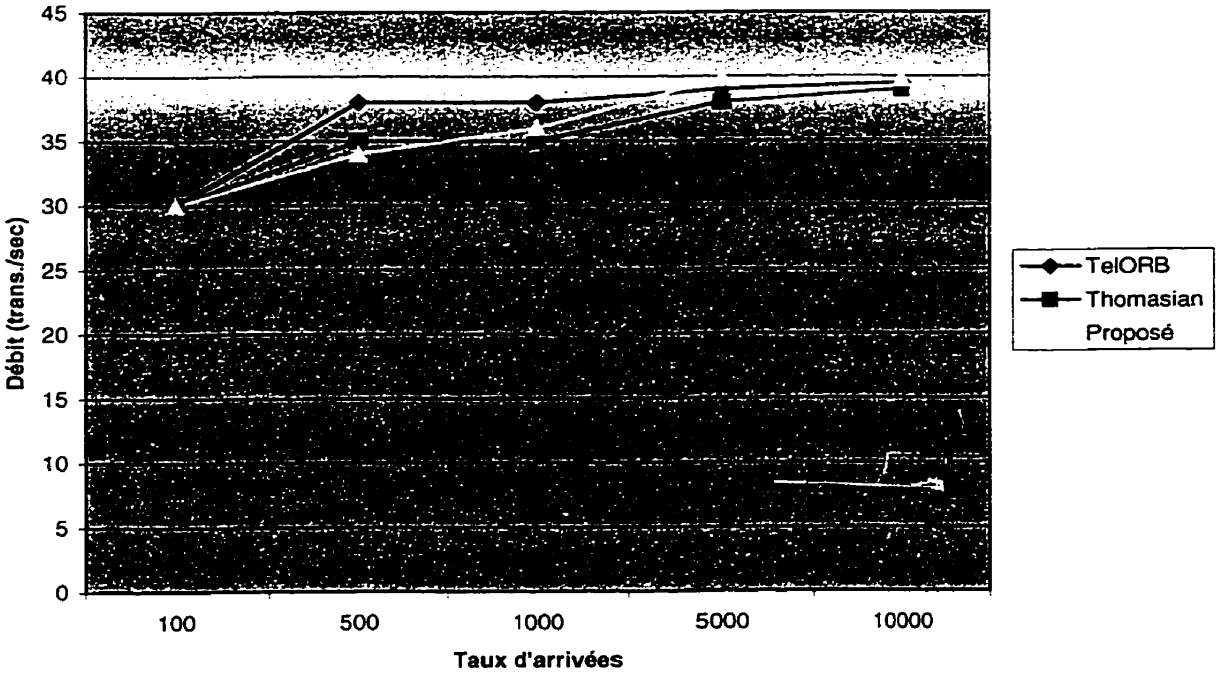


Figure 5.5 Taux de sortie en fonction du taux d'arrivée lorsque la BD est sur disque

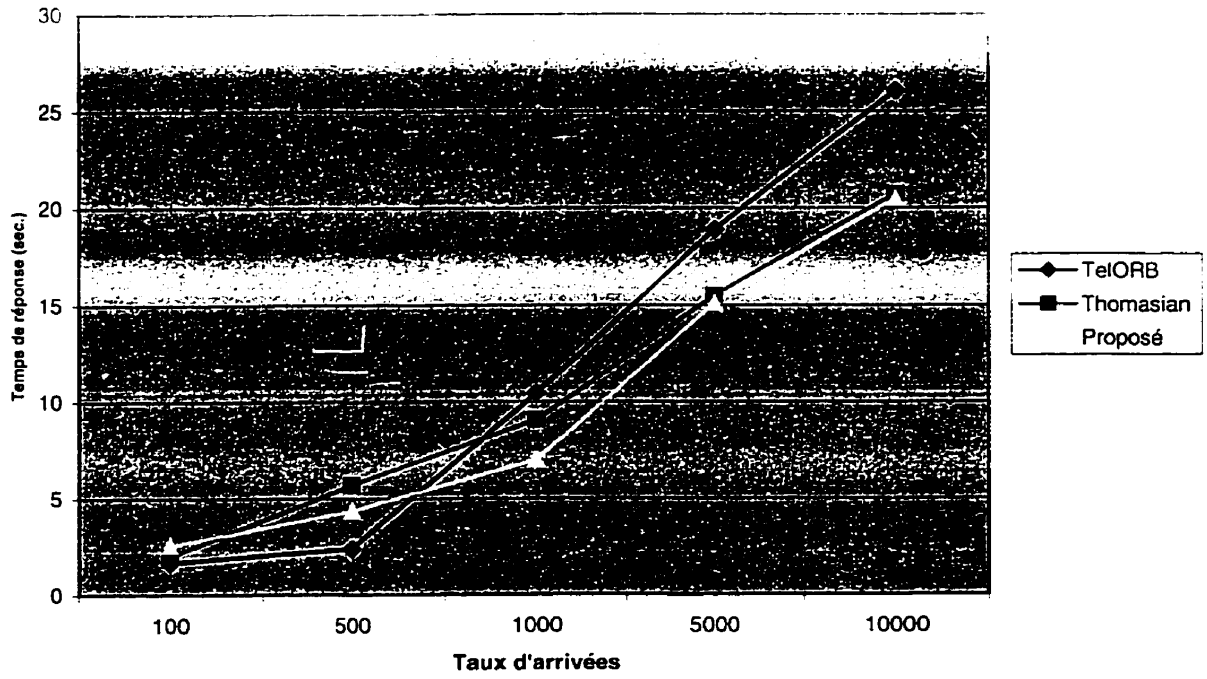


Figure 5.6 Temps de réponse en fonction du taux d'arrivée lorsque la BD est sur disque

À partir d'un certain seuil, le temps de réponse observé par chacun des trois algorithmes augmente plus vite par rapport aux résultats obtenus lorsque la BDR était supposée en mémoire vive. Cette augmentation dans le cas de l'algorithme de TelORB est plus important par rapport aux deux autres algorithmes. Cette situation serait une conséquence de l'importance de la durée du temps de service qui est devenue plus importante. Ces résultats nous amènent à conclure que, pour une BDR localisée sur disque, les performances sont meilleures en utilisant l'algorithme optimiste de Thomasian ou l'algorithme que nous proposons.

5.4 Comportement d'un SGBDR avec réplication de données

Dans cette simulation, chaque entité de donnée dans la BDR existe en deux répliquas. Une opération de lecture est faite sur un seul des répliquas, alors qu'une

opération d'écriture est faite de façon atomique sur les deux répliquas. Nous supposons une situation de conflits possibles et qu'il n'y a pas de contrainte de temps réel. Nous varions le nombre d'arrivées par seconde (NARS) entre 100 et 10 000 et les paramètres fixés prennent les valeurs suivantes :

Svtmp :	temps de service pour une lecture	=	1.0 sec
	temps de service pour une écriture	=	1.5 sec
Iatmp :	temps interarrivée	=	1.0 sec
Nb_sites :	nombre de sites	=	8
Nb_obj :	nombre d'objets/sites	=	100
L_prob :	taux des opérations de lecture	=	5/6
E_prob :	taux des opérations d'écriture	=	1/6
Taille d'une transaction		=	2 opérations

Les résultats de simulation obtenus sont représentés aux figures 5.7 et 5.8.

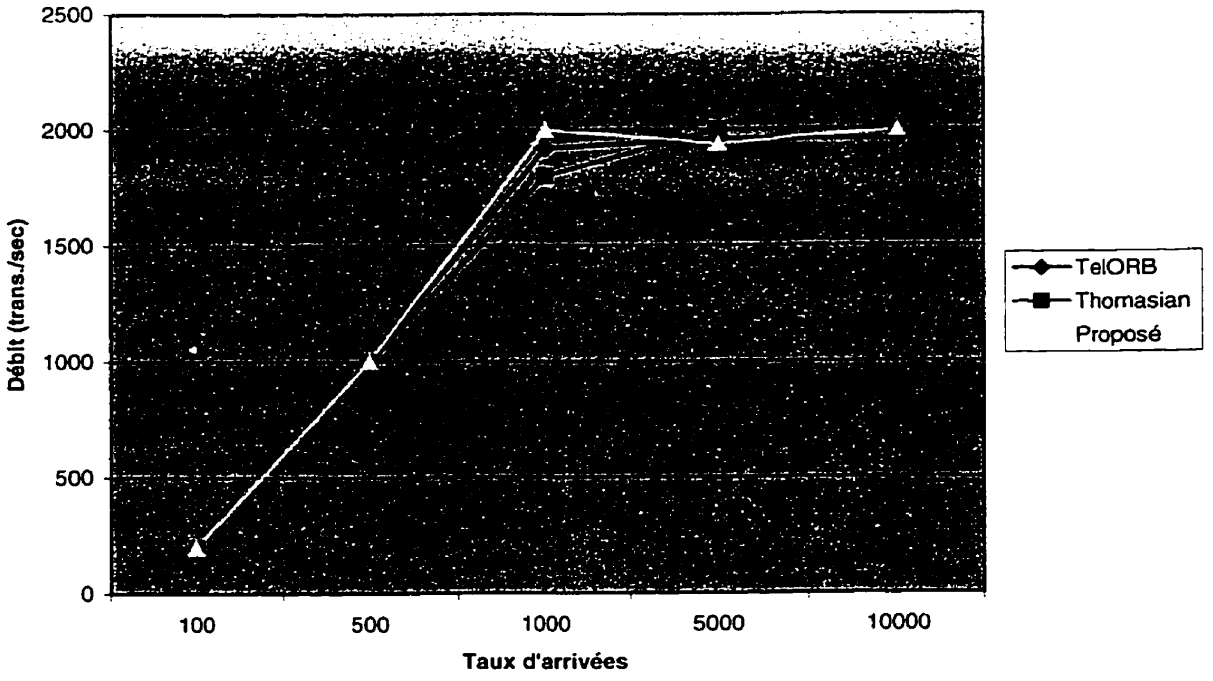


Figure 5.7 Taux de sortie en fonction du taux d'arrivée lorsque la BD est répliquée

Nous pouvons noter qu'au fur et à mesure que l'on augmente le taux d'arrivées, le débit garde presque la même valeur, quel que soit l'ordonnanceur considéré. Le petit décalage observé avec l'algorithme de Thomasian par rapport aux deux autres est négligeable si l'on tient compte de notre marge d'erreur. La courbe du temps de réponse enregistrée avec l'algorithme de TelORB a une pente très faible par rapport aux deux autres. Ces résultats nous amènent à conclure que pour une BDR repliquée, les performances sont meilleures en utilisant l'algorithme de TelORB.

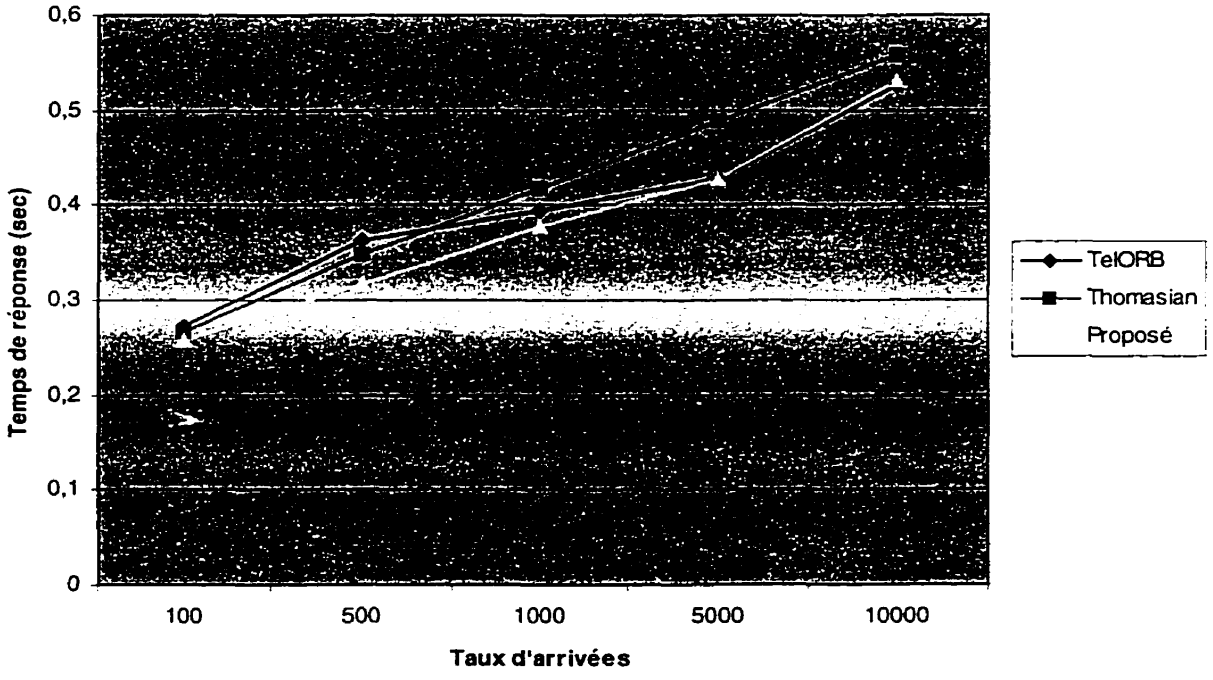


Figure 5.8 Temps de réponse en fonction du taux d'arrivée lorsque la BD est répliquée

5.5 Comportement d'un SGBDR de temps réel

Dans cette simulation, on affecte à chaque opération un délai qui représente la contrainte de temps réel. Lorsque le délai expire, l'opération est réinitialisée. Nous supposons une situation de conflits possibles et qu'il n'y a pas de réplication de

données. Nous varions le nombre d'arrivées par seconde (NARS) entre 100 et 10 000 et les paramètres fixés prennent les valeurs suivantes :

Svtmp :	temps de service pour une lecture	=	1.0 sec
	temps de service pour une écriture	=	1.5 sec
Delai :	2 fois le temps de service	=	3.0 sec
Iatmp :	temps interarrivée	=	1.0 sec
Nb_sites :	nombre de sites	=	8
Nb_obj :	nombre d'objets/sites	=	100
L_prob :	taux des opérations de lecture	=	5/6
E_prob :	taux des opérations d'écriture	=	1/6
Taille d'une transaction		=	2 opérations

Les résultats de simulation obtenus sont représentés aux figures 5.9 et 5.10.

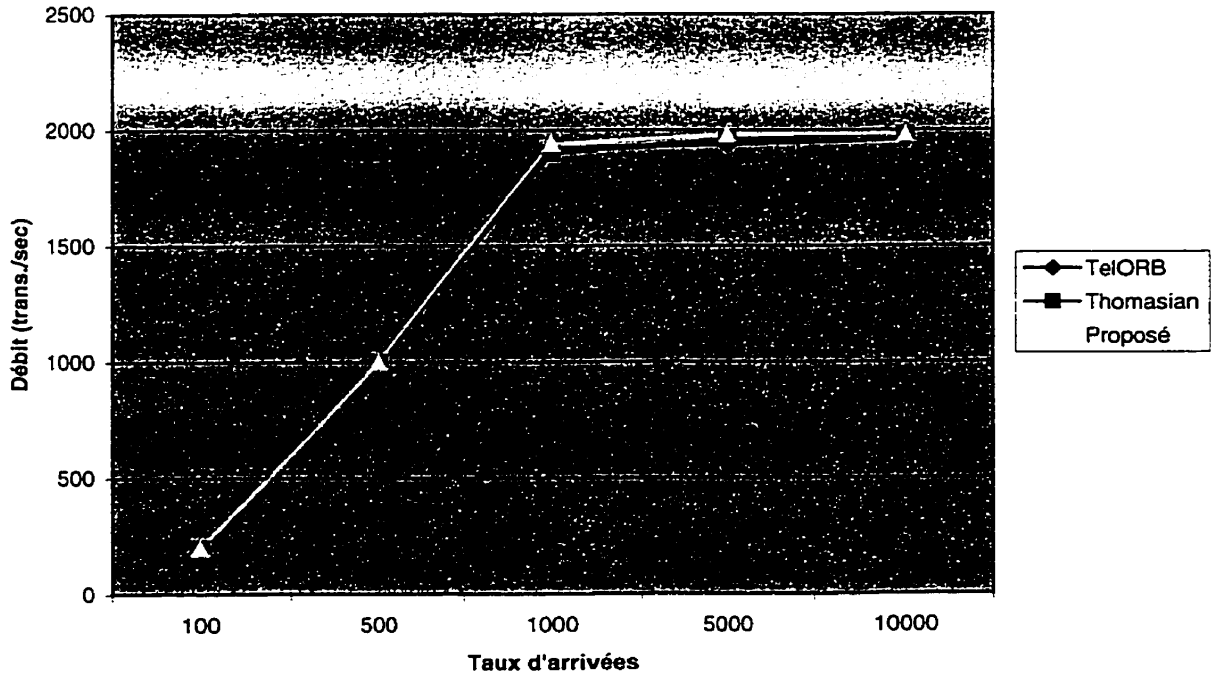


Figure 5.9 Taux de sortie en fonction du taux d'arrivée pour une BD de temps réel

Nous pouvons noter qu'au fur et à mesure que l'on augmente le taux d'arrivée,

les courbes du débit se confondent en tout point pour les trois algorithmes considérés. La courbe du meilleur temps de réponse est enregistrée avec l'algorithme de TelORB. Ces résultats nous amènent une fois de plus à conclure que, pour une BDR temps réel, les performances sont meilleures en utilisant l'algorithme de TelORB.

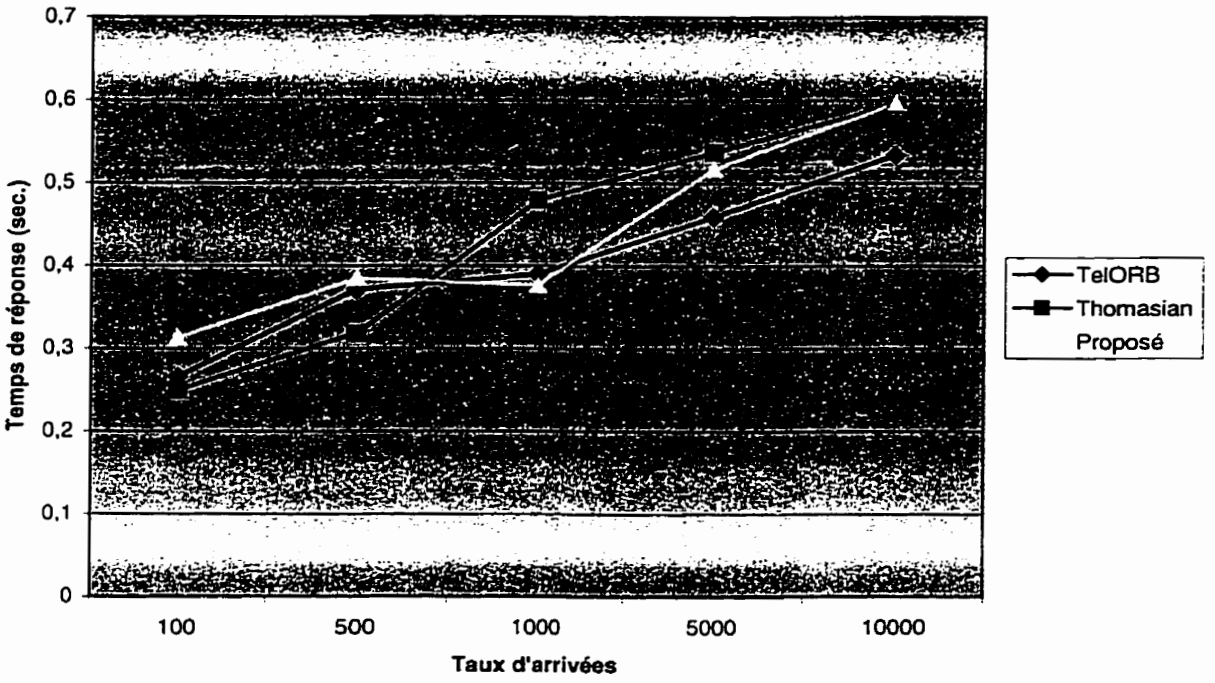


Figure 5.10 Temps de réponse en fonction du taux d'arrivée pour une BD de temps réel

5.6 Synthèse des résultats de simulation

Des expériences réalisées, il ressort que dans une BDR, si la probabilité des accès concurrents est non négligeable, un algorithme d'ordonnancement pessimiste comme celui de TelORB permettrait de garantir une cohérence forte des données, ainsi qu'un niveau de performance acceptable. Cependant, si cette probabilité est plutôt négligeable, l'on peut améliorer davantage la performance du système en utilisant un algorithme optimiste comme celui de Thomasian. L'algorithme optimiste, tel que proposé au départ par Kung et Robinson (1981), offre un degré de performance

inégalable car il suppose que dans une BD. la probabilité des accès concurrents est souvent négligeable; il n'effectue donc aucun ordonnancement au début, mais plutôt des vérifications après exécution. Cependant, s'il advient que cette condition soit fausse, des opérations pourraient être reexécutés plusieurs fois et les performances de cet algorithme se dégraderaient plus vite. Néanmoins, un algorithme optimiste comme celui proposé par Thomasian prévient des reexécutions successives en adoptant plutôt une approche pessimiste lors de la reexécution d'une transaction. On note aussi assez de cas où les performances de l'algorithme de Thomasian sont limitées en raison du fait que sa version originale ne tient compte ni de la réplication, ni de la contrainte de temps réel. On constate que pour chaque schéma de configuration étudié, les performances de l'algorithme que nous proposons sont toujours soit légèrement inférieures, soit égales à celles du meilleur des trois algorithmes.

5.7 Résultats expérimentaux sur DBN

Dans cette section, nous décrivons d'abord le matériel utilisé pour réaliser nos expériences, ensuite l'application que nous avons développée sur DBN (le SGBD de TelORB) et les étapes à suivre pour sa mise en œuvre. Puis, nous présentons les résultats obtenus avant de proposer quelques améliorations que l'on pourrait apporter au système TelORB dans le futur.

5.7.1 Configuration matérielle et logicielle

Nous avons déjà décrit une grappe d'ordinateurs comme étant un ensemble de machines de puissance moyenne, reliées entre elles par un réseau de communications et qui apparaissent à l'usager comme une seule machine à haute performance. Nous avons installé notre application sur une grappe d'ordinateurs ayant les configurations décrites ci-dessous.

Sur le plan matériel :

- 3 processeurs Pentium III de 500 MHz ayant chacun une mémoire RAM de 512 Mo. Un des 3 processeurs a un bus SCSI avec 3 disques de 18 Go et un

DAT DDS-4, alors que les 2 autres sont sans disque

- 1 LAN Ethernet
- 2 commutateurs 3com ayant chacune 37 ports (36 100baseT + 1 fibre)

Sur le plan logiciel :

- le système d'exploitation réparti TelORB version 13,
- Delos et Delux qui font partir de l'environnement de programmation C++ et Java sur TelORB
- le SGBD DBN de TelORB
- le programme interface usager TelORBManager1.0.

5.7.2 Fonctionnement de l'application

Dans l'application, nous distinguons deux types de processus qui s'exécutent en parallèle en communiquant via CORBA. Un processus central de type *statique*, agissant comme générateur de trafic, envoie continuellement des messages à un certain nombre de processus récepteurs de type *dynamique* et qui sont répartis sur les processeurs disponibles dans le système. Une instance de processus *dynamique* représente un usager mobile et est codée comme un *Thread* (un programme qui s'exécute de façon indépendante). Dès qu'il reçoit le message du processus central, le *Thread* va effectuer une ou plusieurs opérations sur un objet dans la base de données avant d'envoyer une réponse au processus central pour signaler la fin des opérations. Nous disposons dans ce programme d'un chronomètre ("*Timer*") qui nous permet de prélever le temps après chaque exécution de 1000 transactions. Cette mesure nous permettra ensuite de déduire le débit ("*throughput*") du système qui est notre premier indice de performance. Pendant que les transactions créées dans l'application manipulent des objets dans DBN, nous comptons le nombre total d'opérations échouées qui nous permettra d'évaluer le taux d'échec qui représente notre deuxième indice performance. Notre programme comprend aussi une interface graphique qui permet l'interaction avec l'usager, à partir d'une station Linux ou Unix. Sur cette interface, l'usager dispose des boutons lui permettant de démarrer ou d'arrêter le programme, et des champs de texte pour entrer les

paramètres de l'expérience à réaliser.

5.7.3 Mise en œuvre de l'application

Les différentes étapes à suivre pour coder et mettre en œuvre une application dans l'environnement de développement de TelORB sont décrites dans Hennert et Larruy (1999). Un processus et le type de données qu'elle peut manipuler sont regroupés ensemble sous un autre type d'objet appelé *Distribution Unit Type (DUT)*. Dans le cas de notre application, il s'agit des objets dans la base de données, c'est-à-dire des objets de type persistant ou *Persitent Object Type (POT)*. Les processeurs disponibles sont regroupés en séries et chaque série est affectée à un DUT. L'avantage d'une telle configuration est que chaque processus a plus de chance d'être exécuté par le processeur dont la mémoire contient la donnée qu'il va manipuler, ce qui minimise les accès distants. L'application comprend toujours plusieurs sous-modules correspondant aux répertoires et sous-répertoires que le programmeur doit créer pour contenir les fichiers de spécification et les codes sources correspondants de l'application. Les interfaces des différents sous-modules d'une application sont enregistrées dans des fichiers de spécification à partir desquels le compilateur pourra générer un squelette de code correspondant, en C++ ou en Java. Ce code ne contient alors que les entêtes des principales procédures et il revient au programmeur de le compléter avec du code spécifique à son application, puis de le compiler. Les prochaines étapes consisteront respectivement à créer les fichiers exécutables, à configurer les variables d'environnement et enfin à exécuter l'application. Rappelons que l'application contient un fichier exécutable par sous-module.

Nous avons créé un objet POT qui contient des informations sur un usager mobile. Le processus central ne manipule aucun objet et constitue le premier DUT qui est associé au processeur numéro 1. Chaque instance de processus dynamique ainsi que le POT correspondant constituent le second DUT auquel sont affectés les processeurs numéro 2 et 3. Lors de la compilation, plus d'une centaine de fichiers de code source ont été générés. Certains de ces fichiers ne nécessitaient plus de modifications à l'instar de

ceux implantant la communication que nous avons spécifiée en CORBA. Nous avons ajouté du code spécifique pour notre application dans les fichiers correspondants au processus statique central ou aux processus dynamiques. Dans la section suivante, nous expliquons les modifications apportées aux fichiers et les résultats des mesures réalisées.

5.7.4 Résultats expérimentaux

Nous avons réalisé un certain nombre d'expériences, dans le but d'évaluer la performance du système TelORB vis-à-vis des applications contenant des transactions moins complexes et ensuite vis-à-vis des applications contenant des transactions plus complexes. Pour cela, nous avons défini dans notre programme un paramètre appelé *Operation*, qui représente le degré de complexité des transactions concurrentes, puis une variable appelée *TailleConcurrence* qui représente le degré de concurrence ou encore le nombre de transactions actives à chaque instant. L'interface usager permet d'entrer des valeurs des deux paramètres du programme avant de démarrer la simulation. Le degré de complexité prend tour à tour les valeurs 0,1,2,3, tandis que nous varions le degré de concurrence entre 10 et 100.

Au démarrage du programme, le processus statique crée un ensemble de 70 000 usagers dans DBN, puis génère un processus dynamique après chaque intervalle de temps correspondant au temps interarrivé. Chaque processus dynamique comprend une transaction qui doit effectuer un certain nombre d'opérations sur les données d'usagers en fonction de la valeur affectée au paramètre *Operation*. Après chaque exécution complète de 1000 transactions, nous mesurons le temps écoulé pour ensuite déduire le débit du système. Les différentes valeurs que nous avons affectées au paramètre *Operation* correspondent aux scénarios suivants :

- 0 : chaque transaction effectue une opération de mise à jour d'un objet choisi au hasard dans DBN.
- 1 : un objet est choisi au hasard et une transaction doit effectuer une opération de création de cet objet si elle n'existe pas déjà dans la base de données, suivie d'une

opération de mise à jour.

- 2 : un objet est choisi au hasard et une transaction doit effectuer d'abord une opération de création de cet objet si elle n'existe pas déjà dans la base de données, suivie d'une opération de mise à jour et enfin d'une opération pour supprimer cet objet.
- 3 : cet exemple un peu complexe illustre une opération bancaire qui consiste à virer un montant d'un compte X à un compte Y. Dans ce cas, le degré de concurrence est plus élevé par rapport aux trois cas précédents, car chaque transaction doit effectuer deux opérations de mise à jour sur deux objets différents.

Les courbes de débit observées pour chacun des trois premiers scénarios sont représentées à la Figure 5.11.

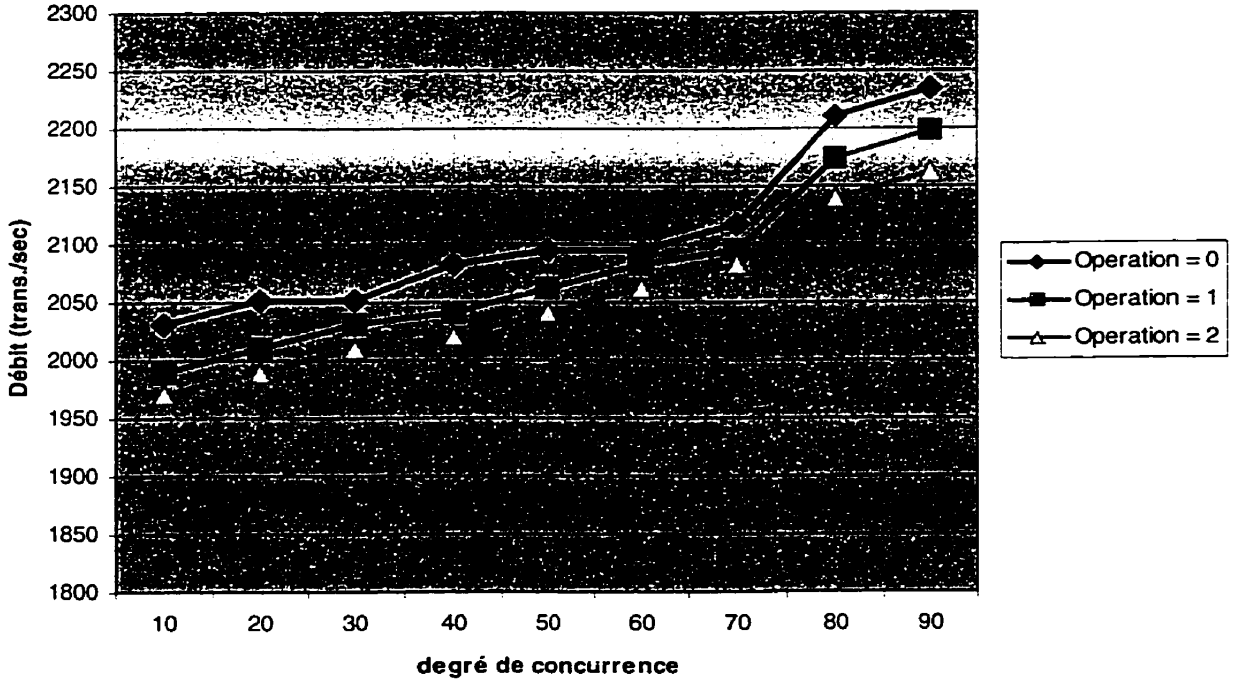


Figure 5.11 Temps de réponse en fonction du degré de concurrence avec variation du degré de complexité des transactions

On peut noter qu'au fur et à mesure que le degré de concurrence augmente, les

trois courbes de débit croissent d'abord de façon linéaire puis évoluent progressivement vers un point de saturation où les processeurs travaillent à leur capacité maximale. Par ailleurs, le débit devient plus faible si le degré de complexité augmente. Les courbes restent parallèles entre elles, ce qui nous permet de conclure que le débit est inversement proportionnel au degré de complexité des transactions. La courbe de débit observée dans le quatrième scénario est représentée à la Figure 5.12.

L'on peut noter que, pour le même degré de concurrence, le débit a diminué considérablement par rapport aux trois courbes précédentes, et que la courbe évolue de façon linéaire.

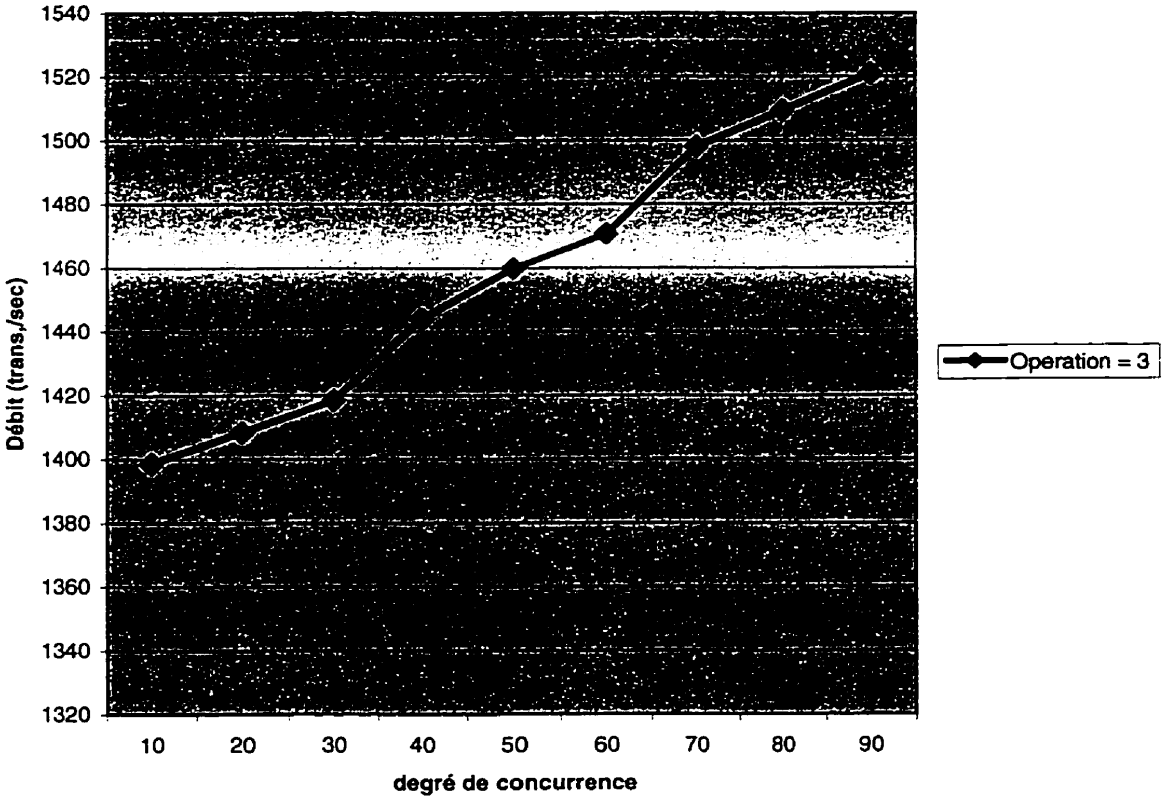


Figure 5.12 Temps de réponse en fonction du degré de concurrence lorsque le degré de complexité est important

Pendant cette expérience, nous avons aussi constaté que toutes les opérations

n'étaient pas complétées comme dans les trois cas précédents. Le taux d'échec mesuré est représenté à la Figure 5.13. Ces échecs observés dans le cas de transactions assez complexes est une conséquence des limitations du mécanisme de contrôle d'accès par verrouillage dont le principe consiste à retarder des opérations de transactions lorsqu'elles sont en conflit. Et dans le cas d'une base de données temps réel comme DBN, une opération doit s'exécuter dans un certain délai au-delà duquel elle est plutôt annulée.

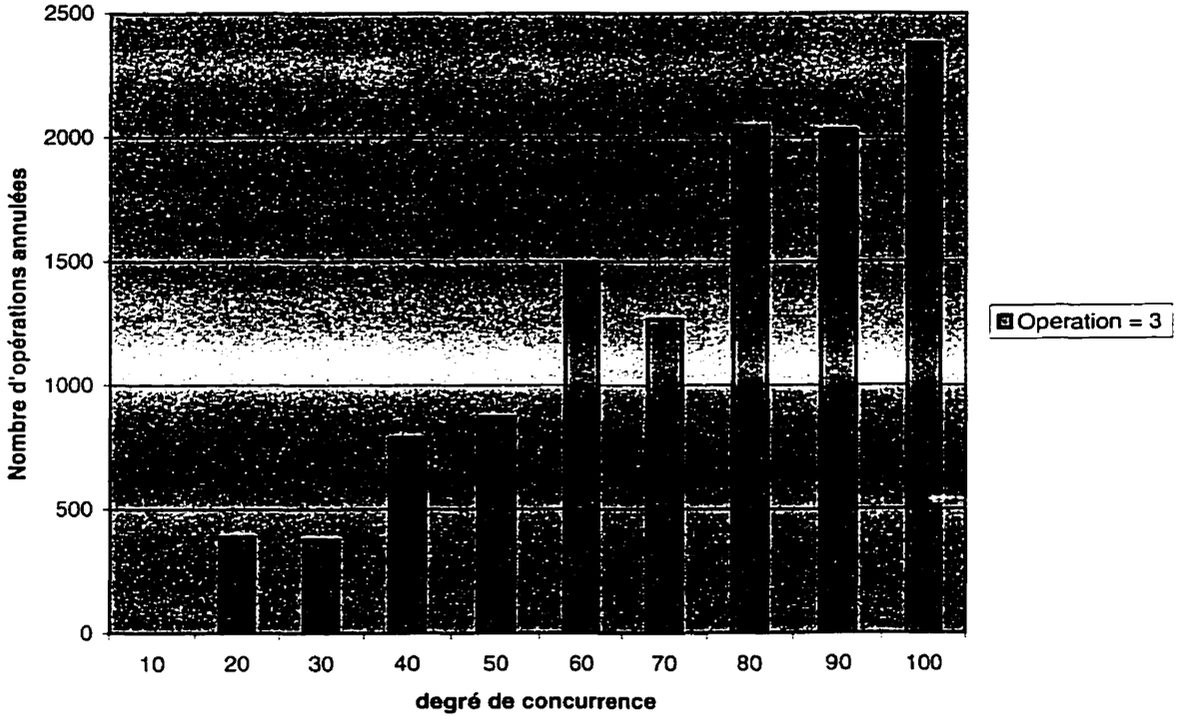


Figure 5.13 Nombre d'opérations annulées en fonction du degré de concurrence

5.8 Améliorations proposées pour DBN

Aujourd'hui, HLR et SCP sont les deux principales applications qui utilisent DBN. Et dans ces applications, les conflits entre transactions sont assez rares car les données d'un usager mobile sont surtout accédées en lecture lorsque le système veut établir un appel. Les seuls cas de mise à jour surviennent seulement lorsqu'un usager est

créé ou lorsqu'il change de zone ou de cellule. Par ailleurs, à cause de la limitation de la bande passante de la liaison SS7 qui relie le système au réseau, le taux d'arrivées des transactions sur DBN reste très faible puisqu'il est fonction de l'intensité du trafic. Les principales exigences des applications HLR et SCP vis à vis de l'algorithme de verrouillage utilisé dans DBN concernent la contrainte de temps réel et la contrainte de haute disponibilité. Dans DBN, la réplication des données et la réduction du délai de possession des verrous sur les répliquas constituent déjà pour les deux applications cibles, un meilleur compromis cohérence-performance.

5.8.1 Améliorations au mécanisme de verrouillage dans DBN

Les expériences réalisées, en particulier lorsque le degré de complexité est égal à 3, nous permettent aussi de conclure que le verrouillage, qui reste encore très utilisé dans la plupart des systèmes de gestion de base de données n'offre pas assez de liberté aux programmeurs d'applications un peu complexe comme celles du commerce électronique. Ces applications qui utilisent très souvent la base de données ont besoin d'un temps de réponse très faible et d'une garantie de cohérence forte pour éviter des erreurs dans les transactions. De plus, la variation du nombre d'usagers concurrents est assez considérable ainsi que la fréquence de mise à jour des données. Les exigences des applications de commerce électronique sont étudiées dans Bhargava A. et Bhargava B. (1999), et nous analysons à titre d'exemple le cas d'un serveur de vente aux enchères.

Un vendeur fait une annonce en créant un objet dans la base qui comprend les caractéristiques suivantes : prix minimum, date de début et date de fin de l'enchère. Chaque acheteur soumet sa mise et peut la surenchérir plus tard. À la date de fin de l'enchère, le produit est vendu au plus offrant. Une telle application pourrait entraîner un degré de concurrence élevé entre des opérations de lecture ou de mise à jour des mises sur un produit alors que chaque usager aimerait avoir instantanément la valeur de la plus récente mise. D'où la nécessité de maximiser le degré de performance et la garantie de cohérence forte. De nouvelles applications pourraient entraîner plus ou moins de conflits entre transactions, tandis que d'autres applications ne sont pas sujettes

à de telles contraintes. Puisque les besoins des applications sont variés et parfois contradictoires, la meilleure solution de contrôle des accès concurrents consisterait à implanter dans le SGBD un mécanisme comme celui que nous proposons et qui est décrit en détail au chapitre 3. En bref, notre solution utilise le principe d'héritage et consiste à implanter les fonctionnalités générales à tout algorithme de contrôle des accès concurrents dans une classe de base, puis de dériver de celle-ci d'autres classes qui implantent chacune un algorithme spécifique. De cette façon, les données partagées par un groupe d'applications seront créées, manipulées et contrôlées par l'algorithme qui offre à ce groupe le meilleur compromis cohérence-performance.

5.8.2 Propositions générales concernant DBN

Plusieurs produits SGBDs orientés-objet ou SGBDs résidant en mémoire vive comme DBN sont utilisés dans les réseaux de télécommunications et la plupart d'entre elles supportent les standards de base proposés par l'ODMG. TimesTen, Objectivity, Phasme, Polyhedra sont des SGBDs qui fournissent aux développeurs d'applications des interfaces standards comme SQL, OQL, ODBC ou JDBC. Certains SGBDs promettent d'offrir dans les prochaines versions des interfaces permettant la conversion des données entre leur format propriétaire et le format XML, ce qui représenterait un grand avantage pour les développeurs qui pourront alors manipuler dans le même programme des données créées par des SGBDs dans leur format propriétaire mais exporter au programme suivant le même format XML. Nous suggérons donc la prise en compte de ces besoins et l'intégration d'interfaces correspondantes dans les prochaines versions de TelORB.

CHAPITRE VI

CONCLUSION

Tout au long de notre étude, nous avons essayé de résoudre le problème de maintien de cohérence dans une BDR en appliquant un nouvel algorithme de gestion des accès concurrents. Les résultats obtenus sont très concluants. Dans ce chapitre final, nous présentons une synthèse générale des travaux avant d'indiquer les limitations de notre méthode et les recherches futures que celles-ci inspirent.

6.1 Synthèse des travaux

Dans une BD, des transactions qui sont exécutées en parallèle peuvent entraîner des incohérences lorsqu'elles accèdent à une même entité de donnée. Le problème de maintien de cohérence dans une BD consiste à implanter dans un SGBD un algorithme d'ordonnancement qui synchronise les accès des transactions concurrentes de manière à préserver l'état cohérent de la base. Une solution triviale à ce problème consiste à ne pas permettre des exécutions en parallèle, mais de placer les transactions concurrentes dans une queue pour ensuite les exécuter les unes après les autres. Cependant, cette solution entraîne un délai d'attente qui conduit à une dégradation de performance lorsque le nombre de transactions en queue devient élevé. La gestion des accès concurrents devient plus complexe dans le cas d'une BDR ou lorsqu'il y a réplication des données.

De nos jours, plusieurs dizaines d'algorithmes d'ordonnancement des transactions ont été développés pour résoudre ce problème, mais on peut les regrouper en deux classes principales : les algorithmes pessimistes comme le V2P, et les algorithmes optimistes. Cependant, les architectes de système adoptent souvent des algorithmes hybrides afin de réaliser le meilleur compromis entre les contraintes de cohérence et les exigences de performance. Dans ce mémoire, nous avons développé un nouvel algorithme d'ordonnancement des accès concurrents dans une BDR en adoptant

l'approche hybride proposé par Graham et Shrivastava (1988) qui réalise une composition de plusieurs types d'algorithmes. L'implémentation de notre méthode consiste à partitionner la BDR pour la répartir sur plusieurs sites dont l'ordonnancement des accès est réalisé localement en utilisant l'algorithme de TelORB ou l'algorithme de Thomasian. Ces deux algorithmes sont plus récents et plus performants que ceux utilisés précédemment par Graham et Shrivastava (1988).

Nous avons conçu un modèle de simulation et effectué une série d'expériences à l'aide du logiciel CSIM en considérant respectivement l'algorithme de TelORB, celui de Thomasian et celui que nous proposons, puis nous avons présenté une analyse comparative des résultats obtenus. Nous avons testé ces trois algorithmes sous plusieurs schémas de configuration : SGBDR en mémoire vive, SGBDR sur disque, SGBDR avec réplication de données, SGBDR de temps réel. En variant le taux d'arrivée des transactions, nous avons mesuré le débit et le temps de réponse moyen. Dans le cas de l'algorithme de TelORB, nous avons refait les mêmes expériences sur le système réel. De manière générale, les résultats observés sont convaincants et pour plusieurs schémas de configuration considérés, nous constatons que l'algorithme que nous proposons offre une meilleure garantie de performance par rapport aux deux autres qui ne tiennent pas compte de la diversité ou des contradictions entre les exigences des applications. Cependant, notre approche n'est pas parfaite et comporte quelques limitations.

6.2 Limitations des travaux

En dépit des résultats satisfaisants obtenus, les performances de notre approche dépendent aussi des hypothèses considérées et du choix des paramètres. En effet, le programme de simulation comprend beaucoup de paramètres, ce qui donne lieu à un grand nombre de combinaisons différentes correspondant chacune à une session d'expériences. Les indices que nous avons utilisés lors de nos expériences pour mesurer la performance des algorithmes sont celles qui sont généralement utilisés par la plupart des chercheurs qui conçoivent les algorithmes de maintien de cohérence. Cependant,

sous des hypothèses propres à d'autres systèmes qui ne sont pas des grappes d'ordinateurs. certains de ces chercheurs analysent aussi d'autres phénomènes que nous n'avons pas considérés dans cette étude. Par exemple, nous avons considéré que la transmission est parfaite sur le réseau de communications qui relie les nœuds de la BDR, ce qui n'est pas toujours le cas. Si les nœuds sont géographiquement éloignés les unes par rapport aux autres, le délai de transmission devient assez important et la fiabilité du réseau n'est plus parfaite.

Notre modélisation de l'environnement TelORB sur Windows pourrait avoir des écarts avec la réalité car, dû au fait que le système est propriétaire, nous n'avons pas trouvé de logiciel de simulation compatible avec cette plate-forme et nous avons alors choisi de coder nos algorithmes avec l'outil de simulation CSIM sur Windows. Cependant, nous avons effectué des expériences sur TelORB et mesuré les performances de l'algorithme de TelORB, mais le temps nécessaire pour se familiariser au système ne nous a pas permis d'implanter aussi l'algorithme optimiste et celui que nous proposons pour ensuite confronter les performances des trois algorithmes sur une plate-forme réelle.

6.3 Indications de recherche future

Le problème de maintien de cohérence dans une BDR demeure un problème ouvert et de nouvelles versions d'algorithmes pessimistes, optimistes ou hybrides pourraient être développées en relaxant certaines contraintes par rapport aux algorithmes existants. On pourrait par exemple essayer d'adapter l'algorithme de Thomasian en supposant cette fois-ci que les données sont répliquées ou que les contraintes de temps réel sont prises en compte. Dans le cas de notre algorithme ainsi que dans celui de l'algorithme de TelORB ou de Thomasian, les recherches futures pourraient aussi essayer de tenir compte de la distance entre les différents sites de la BDR et du délai que ceci pourrait entraîner dans la communication entre les différents modules du SGBDR qui contrôlent localement sur chaque site les accès des transactions aux données. Pour

les expériences futures, il faudrait rendre disponibles les résultats des mesures réalisées sur TelORB/DBN. Enfin, il faudra développer un outil logiciel de simulation compatible avec le système d'exploitation TelORB.

BIBLIOGRAPHIE

Agrawal D., A.J. Bernstein, P. Gupta and S. Sengute, "Distributed Optimistic Concurrency Control with Reduced Rollback", *Distributed Computing*, Vol. 2, No. 1, pp. 45-59, 1987.

Agrawal D., M. J. Carey, and M. Livny, "Concurrency Control Performance Modeling : Alternatives and Implications", *ACM Transactions on Database Systems*, Vol. 12, No. 4, pp. 609-654, Dec. 1987.

Agrawal D., A. El Abbadi and R. Jeffers, "Using Delayed Commitment in Locking Protocols for Real-Time Databases", *ACM SIGMOD*, California USA, pp. 104-113, June 1992.

Bellaïche M., Y. Boudreault, R. Laganière, *Algorithmes et programmation à objets*, Version 2, École Polytechnique de Montréal, 659p, 1998.

Bernstein P. A. and N. Goodman, "A Sophisticate's Introduction to Distributed Database Concurrency Control", *Proceedings of the Eighth International Conference on Very Large Data Bases*, Mexico City, pp. 62-76, September 1982.

Bhargava B., "Concurrency Control in Database Systems", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No 1, pp. 3-16, January/February 1999.

Bhargava A and B. Bhargava, "Measurements and Quality of Service Issues in Electronic Commerce Software", *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering & Technology,* 1999.

Buyya R., *High Performance Cluster Computing*, Volume 1, Architectures and Systems, Prentice Hall, New Jersey, 1999.

Carey M. J. and M. Livny, "Distributed Concurrency Control Performance : A Study of Algorithms, Distribution and Replication", *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, California, pp. 13-25,

1988.

Chiu A., B. Kao and K.-Y. Lam, "Comparing Two-Phase Locking and Optimistic Concurrency Control Protocols in Multiprocessor Real-Time Databases", *IEEE Proceedings of the 1997 Joint Workshop on Parallel and Distributed Real-Time Systems (WPDRTS / OORTS '97)*, pp. 141-148, 1997.

Cingiser L. D. and V. F. Wolfe, "Object-Based Semantic Real-Time Concurrency Control with Bounded Imprecision", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 1, pp. 135-147, January/February 1997.

Cornafion, *Systèmes informatiques répartis, concepts et techniques*, Bordas, Paris, 1981.

Davidson S. B., "Optimism and Consistency in Partitioned Distributed Database Systems", *ACM Transactions on Database Systems*, Vol. 17, No. 3, pp. 456-481, ACM, September 1984.

Dixon G. N. and S. K. Shrivastava, "Exploiting Type-Inheritance Facilities to Implement Recoverability in Object Based Systems", *Proceedings of 6th Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg, pp. 107-114, March 1986.

Eswaran K. P., J.N. Gray, R. A. Lorie and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Relational Database System", *Communications of the ACM*, Vol. 8, No. 11, pp. 624-633, 1976.

Franaszek P. A., J. T. Robinson and A. Thomasian, "Concurrency Control for High Contention Environments", *ACM Transactions on Database Systems*, Vol. 17, No. 2, pp. 304-345, June 1992.

Franaszek P. A., J.R. Haritsa, J. T. Robinson and A. Thomasian, "Distributed Concurrency Control with Limited Waith Depth", *Proc. 12th Int'l Conf. Distributed Computing Systems*, Yokohama, Japan, pp. 160-167, June 1992.

Franaszek P.A. and J.T. Robinson "Limitations of Concurrency in Transaction Processing", *ACM Transactions on Database Systems*, Vol. 10, No. 1, pp. 1-28, March 1985.

Gardarin G., *Bases de Données objet & relationnel*, Éditions Eyrolles, Paris 1999.

Graham D. P. and S. K. Shrivastava, "Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems", *Appeared in the Proceedings of the second European Conference on Object-Oriented Programming*, ECOOP88, Oslo, pp. 1-17, August 1988.

Hennert L., A. Larruy, "TelORB-The distributed communications operating system", *Ericsson Review,* No. 03, 1999.

Hong D., T. Johnson, S. Chakravarthy, "Real-Time Transaction Scheduling : A cost Conscious Approach", *ACM SIGMOD,* Washington, USA, pp. 197-206, May 1993.

Huang J., J. A. Stankovic, K. Ramamritham and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes", *Proceedings of the 17th International Conference on Very Large Databases*, Barcelona, pp. 35-46, September 1991.

Jyhi-Kong W., Y.-S. Hu, C.-H. Chen, and W.-P. Yang , "On the traffic estimation and engineering of GSM network", *IEEE PIMRC,* Vol. 3, pp. 1183-1187, 1996.

Jyhi-Kong W., W.-P. Yang and L.- F. Sun, "Traffic impacts of international roaming on mobile and personal communications with distributed data management", *ACM/Baltzer Mobile Networks and Applications*, Vol. 2, No. 4, pp. 345-356, 1998.

Kersten M. L. and H. Tebra, "Application of an Optimistic Concurrency Control Method", *Software-Practice and Experience*, Vol. 14, No. 2, pp. 153-168, 1984.

Knapp E., "Deadlock Detection in Distributed Databases", *ACM Computing Surveys*, Vol. 1, no. 4, pp. 303-328, Dec. 1987.

Krishnamurthi G., S. Chessa and A.K. Somani, "Fast recovery from database/link failures in mobiles networks", *Computer Communications, Elsevier Science B.V.*, pp. 561-574, 2000.

Kung H.T. and J.T. Robinson, "On Optimistic Method for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 6, No. 2, pp. 213-226 1981.

Lausen G., "Concurrency Control in Database Systems : A Step Towards the Integration of Optimistic Methods and Locking", *Proc. ACM Ann. Conf.*, pp. 64-68, 1982.

Li V., "Performance Model of Timestamp-Ordering Concurrency Control Algorithms in Distributed Databases", *IEEE Transactions on Computer*, C-36, 9, September 1987.

Neal Leavitt, "Whatever Happened to Object-Oriented Databases ?", *IEEE Computer Innovative technology for computer professionals*, Vol. 33, Number 8, pp 16-19, August 2000.

Olson Michael A., "Selecting and Implementing an Embedded Database System", *IEEE Computer Innovative technology for computer professionals*, Vol. 33, Number 9, pp. 27-34, September 2000.

Özsu M. T. and P. Valduriez, *Principles of Distributed Database Systems*, second edition, Prentice Hall, New Jersey, 1999.

Papadimitriou C., *The theory of Database Concurrency Control*, Computer Science Press, 1986.

Pitoura E. and B. Bhargava, "Maintaining Consistency of Data in Mobile Distributed Environments", *Proceedings 15th IEEE International Conference on Distributed Computing Systems*, pp. 404-413, May 1995.

Robinson J., "Design of Concurrency Controls for Transaction Processing Systems", Ph. D. Thesis, Carnegie Mellon University, 1982.

Schwetman Herb, "Object-oriented simulation modeling with C++/CSIM17", *Proceedings of the 1990 Winter Simulation Conference*, Ed. C. Alexopoulos, K. Kang, W. Lilegdon, D. Goldsman, pp. 529 – 533, Washington, D.C., 1990.

Stonebraker M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Transactions on Software Engineering*, SE-5, 3, May 1979.

TelORB/DBN, *DPI Exercise – Database*, 1996.

TelORB/DBN, *DPI Programmer's Guide – Database*, 1996.

TelORB/DBN, *DPI Concepts Manuel – Database*, 1996.

Thomasian A., "On the Number of Remote Sites Accessed in Distributed Transaction Processing", *IEEE Trans. Parallel and Distributed Processing*, Vol. 4, No. 1, pp. 99-103, January 1993.

Thomasian Alexander, "Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No 1, pp. 173-189, January/February 1998.

Ulusoy O., "Processing of Real-Time Transactions in a Replicated Database Systems", *Journal of Distributed and Parallel Databases*, Vol. 2, No. 4, pp. 405-436, 1994.

Wong M.H. and D. Agrawal, "Tolerating Bounded Inconsistency for Increasing Concurrency in Database Systems," *Proc. 11th ACM Symp. Principles of Database Systems (PODS)*, pp. 236-245, 1992.

Zhang A.and Elmagarmid A., "A Theory of Global Concurrency Control in Multidatabase Systems", *VLDB J.*, Vol. 2, No. 3, pp. 331-359, July 1993.

Liens Internet

"CSIM18 User Guide", Mesquite Software, Inc., URL : *http://www.mesquite.com/*

Object-oriented DBMS collection, *http://sal.kachinatech.com/H/2/index.shtml*

Relational DBMS collection, *http://sal.kachinatech.com/H/1/index.shtml*

Little M. C. and D. L. McCue, "Construction and Use of a Simulation Package in C++", *http://arjuna.newcastle.ac.uk/group/abstracts/p078.html*

Muth P., T. C. Rakow, G. Weikum, P. Brössler and C. Hasse, "Semantic concurrency control in distributed object-oriented database systems", *http://ncstrl.gmd.de/Dienst/UI/2.0/Describe/ercim.gmd/ap-ipsi-1993-645?tiposearch=&langver=*